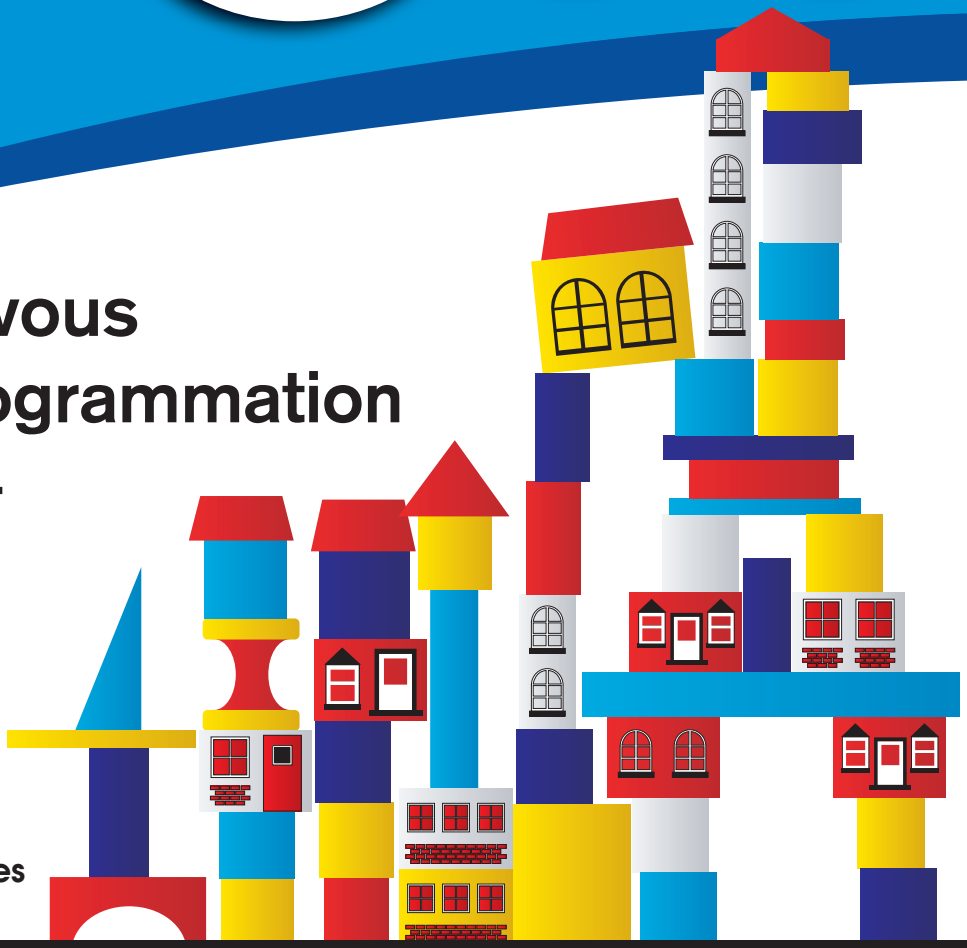
Le Programmeur

Le langage C++

Initiez-vous à la programmation en C++



Jesse Liberty
et Bradley Jones

PEARSON

LE PROGRAMMEUR

Le langage C++

Jesse Liberty et Bradley Jones

PEARSON

Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Mise en pages : TyPAO

ISBN : 978-2-7440-4023-8
Copyright© 2009 Pearson Education France

Tous droits réservés

Titre original : *Teach Yourself C++ in 21 Days, fifth edition*

Traduit de l'américain par Nathalie Le Guillou de Penanros

Nouvelle édition française revue, corrigée et complétée par Eric Jacoboni

ISBN original : 0-672-32711-2
Copyright © 2004 Sams Publishing
All rights reserved

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher.

Sommaire

Introduction 1

Partie I 3
1. Bien débuter en C++ 5
2. Anatomie d'un programme C++ 23
3. Variables et constantes 39
4. Expressions et instructions 65
5. Fonctions 97
6. Programmation orientée objet 135
7. Déroulement d'un programme 173

Partie II 207
8. Pointeurs 209
9. Références 243
10. Fonctions avancées 279
11. Analyse et conception orientée objet 321
12. Héritage 357
13. Tableaux et chaînes 395
14. Polymorphisme 437

Partie III 481
15. Classes et fonctions spéciales 483
16. Concepts avancés d'héritage 515
17. Les flux 575
18. Espaces de noms 619
19. Les modèles 641
20. Gestion des erreurs et exceptions ... 699
21. Et maintenant ? 737

A. Mots-clés C++ 775
B. Priorité des opérateurs 777
C. Solutions des exercices 779
D. Étude des listes chaînées 837

Index 849

Table des matières

Introduction 1
Public visé 1
Conventions typographiques 2

Partie I 3

1. Bien débuter en C++ 5
Introduction 5
Un peu d'histoire... 5
La programmation orientée objet (POO) 9
Évolution de C++ 11
Est-il nécessaire d'apprendre d'abord le langage C ? 11
C++, Java et C# 11
Norme ANSI 12
Préparation à la programmation 13
Votre environnement de développement 13
Création du programme 14
Cycle de développement 15
Votre premier programme C++ : *BONJOUR.cpp* 16
Comment tirer parti de votre compilateur 19
Erreurs de compilation 20
Questions-réponses 21
Testez vos connaissances 22
Exercices 22

2. Anatomie d'un programme C++ 23
Un programme simple 24
Étude rapide de l'objet *cout* 26
Utilisation de l'espace de nom standard 28
Commentaires 30
Les fonctions 33
Questions-réponses 36
Testez vos connaissances 37
Exercices 37

3. Variables et constantes 39
Qu'est-ce qu'une variable ? 39
Définition d'une variable 44
Création de plusieurs variables à la fois 48
Affectation de valeurs aux variables .. 48
Création d'alias avec *typedef* 50
short ou *long* : que choisir ? 51
Variables caractère 54
Constantes 57
Constantes énumérées 59
Questions-réponses 62
Testez vos connaissances 63
Exercices 64

4. Expressions et instructions 65
Instructions 66
Expressions 67
Opérateurs 69
Combinaison d'opérateurs d'affectation et d'opérateurs mathématiques 72
Incrémentation et décrémentation 72
Priorité des opérateurs 75
Parenthèses imbriquées 76
Vrai ou faux 77
L'instruction *if* 78
Utilisation d'accolades dans les instructions *if* imbriquées 86
Opérateurs logiques 89
Évaluation en court-circuit 90
Priorité des opérateurs relationnels 90
Vrai ou faux 91
Opérateur conditionnel (ternaire) 92
Questions-réponses 93
Testez vos connaissances 94
Exercices 95

5. Fonctions 97
Qu'est-ce qu'une fonction ? 98
Valeurs renvoyées, paramètres formels et effectifs 99
Déclaration et définition de fonctions 99
Exécution des fonctions 103
Portée des variables 104
Instructions des fonctions 111
Retour sur les paramètres des fonctions 111
Retour sur les valeurs renvoyées 112
Paramètres par défaut 115
Surcharge de fonctions 117
Pour en savoir plus sur les fonctions .. 121
Principe des fonctions 128
Questions-réponses 132
Testez vos connaissances 133
Exercices 133

6. Programmation orientée objet 135
C++ est-il orienté objet ? 135
Création de nouveaux types 137
Présentation des classes et des membres 138
Accès aux membres d'une classe 141
Accès privé et accès public 142
Implémentations des méthodes de la classe 148
Ajout de constructeurs et de destructeurs 151
Inclure des fonctions membres *const* .. 156
Interface et implémentation 156
Où placer les déclarations de classes et les définitions de méthodes ? 159
Implémentation en ligne 161
Classes imbriquées 163
Les structures 168
Questions-réponses 169
Testez vos connaissances 170
Exercices 171

7. Déroulement d'un programme 173
Les boucles 173
Les boucles *while* 175
Les boucles *do...while* 183
L'instruction *do...while* 184
Les boucles *for* 186
Résumé des boucles 195
Contrôler le flux avec des instructions *switch* 198
Utilisation de *switch* dans un menu ... 201
Questions-réponses 204
Testez vos connaissances 205
Exercices 206

Partie II 207

8. Pointeurs 209
Qu'est-ce qu'un pointeur ? 210
Utilité des pointeurs 220
La pile et l'espace mémoire adressable (tas) 220
En savoir plus sur les fuites mémoire . 225
Création d'objets sur le tas 226
Suppression d'objets du tas 226

Accès aux données membres 228
Création de données membres dans le tas 229
Le pointeur *this* 232
Pointeurs perdus, incontrôlables ou pointeurs fous 233
Pointeurs *const* 237
Questions-réponses 240
Testez vos connaissances 241
Exercices 241

9. Références 243
Qu'est-ce qu'une référence ? 244
Utilisation de l'opérateur *adresse de* (&) sur des références 245
Référencement des objets 248
Pointeurs nuls et références nulles 250
Passage de paramètres par référence .. 251
En-têtes et prototypes de fonctions 256
Renvoi de plusieurs valeurs 257
Efficacité du passage de paramètres par référence 260
Choisir entre références et pointeurs .. 269
Mélanger les références et les pointeurs dans une liste de paramètres 269
Renvoi de références d'objet hors de portée 271
Renvoi d'une référence à un objet dans le tas 272
Où le pointeur est-il passé ? 275
Questions-réponses 275
Testez vos connaissances 276
Exercices 276

10. Fonctions avancées 279
Fonctions membres surchargées 279
Utilisation de valeurs par défaut 282
Choisir entre des valeurs par défaut et des fonctions surchargées 284
Le constructeur par défaut 285
Surcharge des constructeurs 285
Initialisation des objets 287
Le constructeur de copie 288
Surcharge des opérateurs 293
Conversion de type (transtypage) 312
Questions-réponses 317
Testez vos connaissances 318
Exercices 319

11. Analyse et conception orientée objet 321
Les modèles 322
Conception : le langage de modélisation 322
Conception : le processus 324
Étape 1 – La conceptualisation : commencer par la vision 327
Étape 2 – L'analyse : collecter les besoins 327
Étape 3 – La conception 341
Étapes 4 à 6 – Implémentation, tests et déploiement 355
Itérations 355
Questions-réponses 355
Testez vos connaissances 356
Exercices 356

12. Héritage 357
Qu'est-ce que l'héritage ? 357
Privé ou protégé ? 362
L'héritage avec les constructeurs et les destructeurs 365
Redéfinition des méthodes de la classe de base 372
Méthodes virtuelles 378
Questions-réponses 392
Testez vos connaissances 393
Exercices 393

13. Tableaux et chaînes 395
Qu'est-ce qu'un tableau ? 395
Accès aux éléments de tableau 396
Écriture d'éléments hors limite 398
Erreurs d'intervalle 401
Initialisation des tableaux 402
Déclaration de tableaux 402
Tableaux d'objets 404
Tableaux multidimensionnels 406

Initialisation de tableaux à plusieurs dimensions 407
Construire des tableaux de pointeurs . 409
L'arithmétique des pointeurs, un sujet complexe 411
Déclaration de tableaux sur le tas 414
Pointeurs sur tableau et tableaux de pointeurs 415
Pointeurs et noms de tableaux 415
Suppression des tableaux stockés sur le tas 417
Redimensionner les tableaux en cours d'exécution 418
Tableaux de caractères et chaînes 421
Les fonctions *strcpy()* et *strncpy()* 423
La classe *String* 425
Listes chaînées et autres structures 433
Création de classes tableaux 434
Questions-réponses 434
Testez vos connaissances 435
Exercices 436

14. Polymorphisme 437
Problèmes liés à l'héritage simple 437
Héritage multiple 445
Types abstraits de données (TAD) 463
Questions-réponses 477
Testez vos connaissances 478
Exercices 479

Partie III 481

15. Classes et fonctions spéciales 483
Partage d'informations entre objets de même type : données membres statiques 484
Méthodes membres statiques 489
Pointeurs sur des fonctions 492
Pointeurs sur des fonctions membres . 507
Questions-réponses 512
Testez vos connaissances 513
Exercices 513

16. Concepts avancés d'héritage 515
Agrégation 515
Héritage et Délégation/Agrégation 531
Héritage privé 541
Classes amies 550
Fonctions amies 560
Fonctions amies et surcharge d'opérateur 560
Surcharge de *operator<<()* 566
Questions-réponses 570
Testez vos connaissances 571
Exercices 571

17. Les flux 575
Présentation des flux 575
Flux et tampons 578
Les objets E/S standard 579
Redirection des flux standard 579
Les entrées avec *cin* 580
Autres fonctions membres de *cin* 585
Les sorties avec *cout* 594
Manipulateurs, indicateurs d'état et instructions de formatage 597
Flux ou fonction *printf()* 601
Entrées/sorties sur fichier 604
ofstream et *ifstream* 604
Fichiers binaires ou fichiers texte 610
Traitement de la ligne de commande .. 612
Questions-réponses 616
Testez vos connaissances 617
Exercices 617

18. Espaces de noms 619
Utilité des espaces de noms 619
Résolution des fonctions et des classes par leurs noms 620
Création d'un espace de noms 625
Utilisation d'un espace de noms 628
Mot-clé *using* 630
Alias d'espace de noms 634
Espace de noms anonyme 634
L'espace de noms standard *std* 636

Questions-réponses 637
Testez vos connaissances 638
Exercices 639

19. Les modèles 641
Qu'est-ce qu'un modèle 641
Construire une définition de modèle ... 643
Passer des objets modèles instanciés aux fonctions 651
Modèles et amis 652
Utilisation des des modèles 660
La bibliothèque des modèles standard 676
Les conteneurs 676
Les conteneurs séquentiels 677
Les conteneurs associatifs 687
Les classes d'algorithmes 692
Questions-réponses 695
Testez vos connaissances 696
Exercices 697

20. Gestion des erreurs et exceptions ... 699
Erreurs logiques et fautes de syntaxe .. 700
Les coulisses des exceptions 702
Produire ses propres exceptions 706
Créer une classe d'exception 708
Placement des blocs *try* et *catch* 712
Données dans les exceptions et noms des objets exceptions 719
Exceptions et modèles 727
Exceptions et erreurs 730
Recherche des erreurs 731
Questions-réponses 733
Testez vos connaissances 734
Exercices 734

21. Et maintenant ? 737
Le préprocesseur et le compilateur 738
La directive de préprocesseur *#define* . 738
Inclusions et clauses d'inclusion 741
Les macros 742
Manipulation des chaînes 745
Les macros prédéfinies 747
La macro *assert()* 747
Les fonctions *inline* 757
Opérations sur les bits 758
Styles de programmation 765
Commentaires 768
Comment poursuivre votre développement C++ 770
Questions-réponses 772
Testez vos connaissances 773
Exercices 774

A. Mots-clés C++ 775

B. Priorité des opérateurs 777

C. Solutions des exercices 779
Chapitre 1 779
Chapitre 2 780
Chapitre 3 782
Chapitre 4 784
Chapitre 5 784
Chapitre 6 787
Chapitre 7 791
Chapitre 8 793
Chapitre 9 794
Chapitre 10 797
Chapitre 11 802
Chapitre 12 804
Chapitre 13 805
Chapitre 14 806
Chapitre 15 808
Chapitre 16 815
Chapitre 17 819
Chapitre 18 822
Chapitre 19 823
Chapitre 20 828
Chapitre 21 834

D. Étude des listes chaînées 837
Les composants d'une liste chaînée ... 838

Index 849



Introduction

Ce livre a été conçu pour vous initier à la programmation en C++. Vous y apprendrez les principaux concepts du langage (gestion des entrées-sorties, boucles et tableaux, modèles, etc.) et de la programmation orientée objet. Chaque chapitre contient des exemples de programmes qui mettent en œuvre les concepts étudiés et qui sont analysés en détail.

Tous les chapitres se terminent par un jeu de questions-réponses, un quiz et des exercices dont vous trouverez les réponses à l'Annexe D.

Les listings de code de chacun des chapitres sont disponibles sur le site **www.pearson.fr**, à la page consacrée à cet ouvrage.

Public visé

Il n'est pas nécessaire d'avoir une expérience préalable de la programmation pour tirer parti de ce livre car il part de zéro et vous apprend à la fois le langage et les concepts de programmation C++. Les nombreux exemples de syntaxe et les analyses détaillées des extraits de code constituent un excellent guide touristique pour votre voyage dans cet environnement si riche. Que vous soyez un programmeur débutant ou expérimenté, vous pourrez constater que cet ouvrage vous permettra d'apprendre rapidement et simplement le langage C++.

Conventions typographiques



Ces paragraphes soulignent des informations qui amélioreront l'efficacité de vos programmes C++.



Ces notes fournissent des précisions sur le sujet abordé.



Ces avertissements vous évitent de tomber dans les pièges les plus communs qui peuvent vous gâcher l'existence.

Les encadrés

Ils précisent la syntaxe d'une instruction.

Il peut aussi s'agir d'apartés destinés à attirer votre attention sur un cas particulier ou une situation exceptionnelle.

Faire	*Ne pas faire*
● Un rappel des conseils à suivre.	● Les erreurs ou confusions à éviter.

Cet ouvrage utilise également des polices de caractères différentes pour distinguer le code C++ du texte classique :

- Les commandes, variables et autres éléments de code apparaissent dans le texte dans une `police à chasse fixe` spéciale.
- Les termes nouveaux ou importants sont écrits en *italique*.
- Les parties variables dans les descriptions de syntaxe apparaissent en *`police à chasse fixe italique`*. Ce style signale que vous devez remplacer l'espace réservé par le nom réel du fichier, du paramètre ou de tout autre élément qu'il représente.

En outre, toutes les lignes de code sont numérotées. Vous constaterez parfois que certaines lignes trop longues pour le format de ce livre ont été découpées en plusieurs lignes : en ce cas, les lignes de continuation ne seront pas numérotées et vous devrez saisir ces lignes comme une seule ligne dans votre éditeur de texte.

Partie I

Dans les sept chapitres qui constituent cette partie, vous ferez vos premiers pas en programmation en général et en C++ en particulier. Les Chapitres 1 et 2 abordent les notions de base de la programmation et le déroulement d'une application. Au Chapitre 3, vous apprendrez à gérer des variables et des constantes dans un programme. Le Chapitre 4 est consacré aux branchements conditionnels réalisés à l'aide d'instructions et d'expressions. Vous vous initierez, au Chapitre 5, à l'utilisation des fonctions, et à celle des classes et des objets au Chapitre 6. Enfin, à partir du Chapitre 7, vous serez en mesure de comprendre le déroulement d'une application et d'écrire vos premiers programmes orientés objet.



1

Bien débuter en C++

Au sommaire de ce chapitre

- Pourquoi choisir le langage C++ ?
- Les étapes du cycle de développement d'un programme
- Écrire, compiler et lancer votre premier programme C++

Introduction

Bienvenue dans *Le Langage C++* de la collection Le Programmeur ! Cet ouvrage a pour objectif de vous initier efficacement à la programmation en langage C++.

Un peu d'histoire...

Les langages de programmation ont considérablement évolué depuis les premiers calculateurs qui avaient été conçus pour les calculs de trajectoires d'artillerie durant la seconde guerre mondiale. À cette époque, les programmeurs travaillaient avec le langage informatique le plus primitif qui soit – le langage machine – ce qui les obligeait à gérer de longues

chaînes de 1 et de 0. Puis, les premiers assembleurs apparurent, afin de rendre les instructions machine plus compréhensibles et plus faciles à utiliser puisqu'elles étaient désormais représentées par des instructions mnémoniques comme `MOV` et `ADD`.

Dans les années 60 apparurent des langages plus évolués comme BASIC et COBOL, qui permettaient aux programmeurs d'utiliser une syntaxe proche de la langue anglaise (le code source), avec des instructions et des mots comme `Let I = 100`. Les lignes d'instructions étaient ensuite traduites en langage machine par des interpréteurs ou des compilateurs.

Un *interpréteur* traduit et exécute une à une les instructions du programme (ou *code source*) et les transforme directement en actions.

Un *compilateur* passe par une étape intermédiaire (la compilation) qui produit d'abord un fichier objet. Le compilateur fait ensuite appel à un éditeur de liens (ou *linker*) qui transforme ce fichier objet en programme exécutable.

Les interpréteurs lisent les instructions ligne à ligne et exécutent le code immédiatement, ce qui en simplifie l'utilisation. Aujourd'hui, les programmes interprétés sont généralement appelés *scripts*.

Certains langages, comme Visual Basic 6, désignent l'interpréteur sous le nom de *bibliothèque d'exécution*. D'autres langages, comme Visual Basic .NET et Java, disposent d'un autre composant, appelé "machine virtuelle" (VM, *Virtual Machine*) ou exécuteur. Bien que la VM soit aussi un interpréteur, il ne s'agit plus d'un interpréteur de code source qui se contente de traduire un langage lisible en code objet : la VM interprète et exécute un "langage machine indépendant de l'ordinateur" compilé, parfois appelé "langage intermédiaire".

Les compilateurs ajoutent les étapes supplémentaires de compilation du code source (compréhensible par l'homme) en code objet (lisible par la machine). Ceci pourrait apparaître comme un inconvénient mais, en réalité, cette étape permet de créer un programme dont la vitesse d'exécution est optimisée puisque la traduction du fichier source en langage machine a déjà été réalisé une fois pour toutes, au moment de la compilation. Il n'est donc plus nécessaire de retraduire le programme à chaque fois qu'on l'exécute.

L'autre avantage des langages compilés comme C++ tient à la diffusion des programmes puisque l'on peut distribuer un fichier exécutable à des personnes qui ne disposent pas du compilateur. Avec un langage interprété, par contre, l'utilisateur doit nécessairement posséder l'interpréteur pour pouvoir exécuter le programme.

C++ est généralement un langage compilé, même s'il existe quelques interpréteurs C++. À l'instar de nombreux langages compilés, il a la réputation de produire des programmes rapides et performants.

En fait, pendant longtemps, la principale préoccupation des programmeurs était de concevoir des applications très courtes pouvant s'exécuter rapidement, car la mémoire et le temps de calcul coûtaient cher. Avec la miniaturisation des ordinateurs, l'augmentation de

leurs performances et la chute des prix, les priorités ont changé. Désormais, le coût de développement dépasse largement celui d'un ordinateur de type PC ou mini. L'important est d'écrire des programmes performants, bien construits et faciles à mettre à jour (c'est-à-dire sans surcoût excessif).



Le mot programme a deux sens. Il désigne les instructions (ou code source) écrites par un développeur, mais également l'ensemble d'un logiciel exécutable. Cette homonymie peut être une source de confusion et il est important de faire la distinction entre le fichier source et le programme exécutable.

Comment résoudre les problèmes

Les problèmes auxquels sont confrontés les programmeurs sont totalement différents de ceux qu'ils devaient résoudre il y a une vingtaine d'années. Dans les années 80, les programmes traitaient des volumes importants de données brutes. Le programmeur et l'utilisateur final étaient tous deux des spécialistes de l'informatique. De nos jours, les utilisateurs sont bien plus nombreux et peu connaissent tous les détails des ordinateurs et du fonctionnement des programmes. L'utilisateur final actuel recherche des solutions prêtes à l'emploi et capables d'effectuer des opérations de gestion courantes ou ponctuelles.

L'informatique est devenue plus conviviale, mais ce processus a également conduit à la mise en œuvre de programmes de plus en plus complexes. Dans les années 70, les utilisateurs étaient contraints de saisir des commandes énigmatiques pour voir défiler à l'écran des volumes impressionnants de données brutes. Cette époque est révolue ! Une application se compose désormais de fenêtres, de menus et de boîtes de dialogue intégrés à une interface conviviale.

Avec le développement du Web, les ordinateurs ont abordé une ère nouvelle de pénétration du marché ; les utilisateurs d'ordinateurs sont plus nombreux que jamais, et ils sont très exigeants.

Au cours des dernières années, les applications se sont également étendues à d'autres périphériques : l'ordinateur de bureau n'est plus la seule cible des applications. Les téléphones portables, les assistants personnels (PDA), les PC de poche et autres périphériques constituent des cibles toutes trouvées pour les applications modernes.

Depuis la première édition de ce livre, les programmeurs ont répondu aux demandes des utilisateurs et les programmes sont devenus plus volumineux et plus complexes. La nécessité de développer des techniques de programmation permettant de gérer cette complexité est devenue évidente.

Les besoins changeant, les techniques et les langages évoluent également pour aider les programmeurs à gérer la complexité des demandes. Dans cet ouvrage, nous nous

concentrerons uniquement sur une partie essentielle de cette évolution : le passage de la programmation procédurale à la programmation orientée objet.

La programmation procédurale, structurée et orientée objet

Il y a quelques années encore, les programmes étaient conçus comme des suites de procédures destinées à traiter les données. Une *procédure* – également appelée *fonction* ou *méthode* – est un ensemble d'instructions s'exécutant l'une après l'autre. Les données et les procédures étaient totalement dissociées et le travail du programmeur consistait à connaître les fonctions appelées par d'autres fonctions et les données qui étaient modifiées. Pour faire face à ce niveau de complexité, on a donc inventé la *programmation structurée*.

Le principe général de la programmation structurée consiste à diviser pour mieux régner. Un programme peut alors être considéré comme un ensemble de tâches. Toute opération trop complexe pour être décrite simplement est décomposée en un ensemble d'opérations plus simples, jusqu'à n'obtenir que des tâches suffisamment triviales pour être aisément compréhensibles.

Le calcul du salaire moyen de chaque employé d'une entreprise est, par exemple, une tâche assez complexe. Toutefois, il est possible de diviser le traitement en plusieurs tâches secondaires :

1. Compter le nombre d'employés.
2. Déterminer le revenu de chaque employé.
3. Faire le total de tous les salaires.
4. Diviser cette valeur par le nombre d'employés.

La troisième étape (total des salaires) peut également se diviser en tâches plus simples :

1. Lire l'enregistrement de chaque salarié.
2. Extraire le salaire de chaque employé.
3. Ajouter cette valeur au total général.
4. Accéder à l'enregistrement suivant.

La lecture de chaque enregistrement peut, de la même façon, être décomposée en opérations plus élémentaires :

1. Ouvrir le fichier des employés.
2. Rechercher le bon enregistrement.
3. Lire les données.

La programmation structurée résout des problèmes complexes de manière très fiable. Toutefois, à la fin des années 80, cette méthode a montré ses limites.

D'une part, il est naturel d'associer les données (les enregistrements des employés, par exemple) et leur traitement (tri, modification, etc.). Malheureusement, la programmation structurée sépare les données des fonctions qui les manipulent et ne propose pas de moyen naturel de les regrouper. La programmation structurée est donc souvent désignée par le terme de *programmation procédurale*, car elle met l'accent sur les procédures (plutôt que sur les *objets*).

D'autre part, les programmeurs devaient souvent réutiliser des fonctions. Or, certaines fonctions qui convenaient à un type de données ne pouvaient pas toujours être réutilisées avec d'autres, ce qui limitait leurs avantages.

La programmation orientée objet (POO)

La programmation orientée objet (POO) répond à ces besoins. Elle fournit les techniques permettant de traiter des applications très complexes, exploite des composants logiciels réutilisables et associe les données aux tâches qui les manipulent.

La caractéristique essentielle de la programmation *orientée objet* est de modéliser des "objets" (c'est-à-dire des concepts) plutôt que des "données". Ces objets peuvent être des éléments graphiques affichables, comme des boutons ou des zones de liste, ou des objets réels, comme des clients, des bicyclettes, des avions, des chats ou de l'eau.

Les objets possèdent des caractéristiques, également appelées propriétés ou attributs, comme âge, rapidité, volume, noir, humide. Ils ont aussi des fonctionnalités, appelées opérations ou fonctions, comme accélérer, voler, miauler ou couler. Le rôle de la programmation orientée objet est de représenter ces objets dans le langage de programmation.

C++ et la programmation orientée objet

Le langage C++ permet d'utiliser toutes les possibilités de la programmation orientée objet, notamment ses trois piliers que sont l'encapsulation, l'héritage et le polymorphisme.

Encapsulation

Un technicien ne fabrique pas les composants qu'il assemble. Il les choisit selon leurs spécifications sans se préoccuper de leur fonctionnement interne.

L'autonomie d'un objet est une propriété résultant d'un processus appelé *encapsulation*. Celle-ci permet de masquer les données internes d'un objet et de l'utiliser sans connaître les détails de son fonctionnement, exactement comme vous utilisez votre réfrigérateur

sans comprendre le principe des compresseurs. Il reste possible de modifier ce fonctionnement interne sans affecter celui du programme, à condition toutefois que les spécifications soient respectées (le compresseur du réfrigérateur peut être remplacé par un autre de conception similaire).

Lorsque notre technicien veut utiliser un composant électronique, il n'a pas besoin d'en connaître les rouages internes. Toutes les propriétés de ce composant sont encapsulées dans l'objet composant, elles ne sont pas communiquées au circuit monté. Il n'est pas nécessaire de connaître le fonctionnement du composant pour l'utiliser efficacement. Ce fonctionnement est masqué par son boîtier.

Le langage C++ gère l'encapsulation à l'aide de types définis par l'utilisateur : les *classes*. Pour en savoir plus sur la conception d'une classe, reportez-vous au Chapitre 6. Si elle est correctement définie, une classe agit comme une entité encapsulée – elle fonctionne comme un composant autonome. Comme pour un objet du monde réel, son fonctionnement interne peut être masqué. Les utilisateurs d'une classe bien conçue n'ont pas besoin de savoir comment elle fonctionne, mais uniquement comment l'utiliser.

Héritage et réutilisabilité

Lorsqu'ils souhaitent concevoir une nouvelle voiture, les ingénieurs de Superauto ont le choix entre monter un projet à partir de zéro et modifier un modèle existant. Le modèle Pipo est peut-être parfait pour la ville, mais cette voiture manque de nervosité sur autoroute. Les ingénieurs ont donc décidé de lui adjoindre un turbo-compresseur et une boîte à six vitesses. Le responsable du projet préfère donc partir d'un modèle existant, le perfectionner, le tester, et l'appeler Star. La Star sera donc une sorte de Pipo, mais il s'agira d'une version spéciale, disposant de nouvelles fonctionnalités.

Le langage C++ implémente la notion d'héritage. Grâce à l'héritage, vous pouvez déclarer un nouveau type à partir d'un type existant. On dit que la sous-classe obtenue dérive du type existant ; on la nomme quelquefois "type dérivé". Si la Star est dérivée de la Pipo et hérite donc de toutes ses qualités, certaines propriétés peuvent lui être ajoutées, d'autres modifiées. Pour en savoir plus sur l'héritage et son application en C++, reportez-vous aux Chapitres 12 et 16.

Polymorphisme

La Star ne répondra pas nécessairement de la même façon que la Pipo à un appui sur l'accélérateur. En effet, la première peut utiliser son injection et son turbo alors que la seconde devra se contenter de sa carburation classique. Quoi qu'il en soit, il suffit au conducteur de démarrer son véhicule et de se déplacer où bon lui semble. Il n'est pas obligé de connaître les détails du moteur.

En C++, des objets différents peuvent avoir des comportements adéquats différents en réponse à la même action grâce au polymorphisme de fonction et de classe. *Poly* signifie plusieurs et *morphe* signifie forme. Le polymorphisme se traduit donc par un nom unique pouvant prendre plusieurs formes, et il est traité aux Chapitres 10 et 14.

Évolution de C++

Lorsque les qualités de la programmation, de la conception et de l'analyse orientées objet commencèrent à être reconnues, Bjarne Stroustrup créa C++ à partir du langage le plus utilisé pour le développement des logiciels professionnels, le langage C. Il lui ajouta tous les éléments nécessaires à la programmation orientée objet.

On a coutume de dire que C++ est un surensemble de langage C et que, par conséquent, tout programme C est virtuellement un programme C++. Pourtant, ces deux langages sont très différents. Pendant longtemps, C++ a attiré les programmeurs C car ces derniers trouvaient sa syntaxe familière. Toutefois, pour tirer le meilleur profit des fonctionnalités de C++, de nombreux développeurs ont compris qu'ils devaient laisser de côté une partie de leurs acquis en C et aborder les problèmes différemment.

Est-il nécessaire d'apprendre d'abord le langage C ?

Nombreux parmi vous sont ceux qui vont se poser cette question, puisque C++ est un *surensemble* du langage C. La réponse de son créateur et de la plupart des programmeurs C++ est la suivante : il est inutile d'apprendre le langage C, voire préférable de commencer directement par le langage C++.

La programmation C est fondée sur les concepts de programmation structurée, alors que la programmation C++ repose sur ceux de la programmation orientée objet. Si vous apprenez d'abord le langage C, vous devrez alors vous défaire des habitudes néfastes liées à ce langage.

Ce livre ne s'adresse pas obligatoirement à un public ayant une expérience préalable de la programmation. Si vous êtes programmeur C, vous pouvez vous contenter de survoler les premiers chapitres du livre. Le développement orienté objet n'est réellement abordé qu'à partir du Chapitre 6.

C++, Java et C#

C++ est l'un des principaux langages pour le développement de logiciels professionnels. Ces dernières années, Java a eu un temps la faveur des programmeurs. Toutefois, nombre

d'entre eux, qui avaient abandonné C++ au profit de Java, ont depuis fait marche arrière. En tout état de cause, les deux langages sont si semblables qu'apprendre l'un revient à connaître 90 % de l'autre.

C# (prononcez *C sharp*) est un langage plus récent développé par Microsoft pour la plate-forme .NET. Il utilise la même syntaxe que C++ et, bien que ces deux langages diffèrent en quelques points importants, l'apprentissage de C++ apporte la majorité des connaissances nécessaires à l'utilisation de C#. Si vous décidiez par la suite d'apprendre C#, l'investissement réalisé dans l'apprentissage de C++ vous sera très bénéfique.

Extensions gérées de Microsoft pour C++

Avec l'arrivée de .NET, Microsoft a introduit les extensions gérées pour C++ (*Managed C++*). Il s'agit d'une extension du langage C++ lui permettant d'utiliser la nouvelle plate-forme Microsoft et ses bibliothèques. Managed C++ permet surtout à un programmeur C++ de profiter des fonctionnalités avancées de l'environnement .NET. Au cas où vous décideriez de créer des applications spécifiquement conçues pour la plate-forme .NET, vous devrez étendre votre connaissance du C++ standard pour y inclure ces extensions.

Norme ANSI

Le comité d'accréditation des standard, qui dépend de l'ANSI (*American National Standards Institute*), a élaboré un standard international pour C++.

Le standard C++ est également appelé norme ISO (*International Organization for Standardization*), norme NCITS (*National Committee for Information Technology Standards*), norme X3 (ancienne appellation de NCITS), ou norme ANSI/ISO. Dans ce livre, nous continuerons à faire référence à la norme ANSI car c'est le terme le plus utilisé.

L'objectif du standard ANSI est de garantir la portabilité du C++ afin que le code que vous allez écrire pour le compilateur de Microsoft, par exemple, ne génère pas d'erreur avec un autre compilateur. Le code présenté dans ce livre étant compatible ANSI, il pourra être compilé sans erreur sur des plates-formes Macintosh, Windows ou Unix.

Pour la plupart des utilisateurs C++, le standard ANSI est transparent. La version la plus récente de cette norme est l'ISO/IEC 14882-2003. La version précédente, l'ISO 14882-1998, a bénéficié d'une stabilité durable et tous les fournisseurs de renom la prennent en charge. Nous nous sommes efforcés d'assurer une totale compatibilité ANSI du code dans cette édition.

Toutefois, n'oubliez pas que les compilateurs ne sont pas tous totalement compatibles avec la norme. En outre, certaines parties de la norme ont été laissées au choix du concepteur

du compilateur : il n'est donc pas garanti que ces parties se compilent ou fonctionnent de la même manière avec des compilateurs différents.



Les extensions gérées pour C++ ne s'appliquant qu'à la plate-forme .NET et, ne faisant pas partie de la norme ANSI, elles ne sont pas traitées dans cet ouvrage.

Préparation à la programmation

Plus que tout autre langage, C++ oblige le développeur à concevoir soigneusement une application avant de l'écrire. Les programmes figurant dans les premiers chapitres de cet ouvrage ne nécessitent pas d'analyse car ils sont triviaux. En revanche, les problèmes complexes rencontrés en programmation professionnelle ne peuvent être résolus qu'avec cette approche. La conception et l'analyse permettent de mieux cerner les différents aspects du problème. Un programme bien construit ne contient pas d'erreurs et peut aisément être mis à jour. D'après des études récentes, on estime que le coût d'un programme repose à 90 % sur la mise au point et la maintenance. La phase de conception permet de réduire les coûts et, par là même, le prix de revient du logiciel.

Avant de passer à la conception d'un programme, vous devez connaître parfaitement le problème à résoudre. Les programmes les plus simples comme les plus complexes s'articulent autour d'un déroulement clair et logique.

Il convient également de déterminer si le problème peut être résolu à l'aide d'un programme existant qui sera modifié, ou à l'aide d'un logiciel du commerce. Qu'il choisisse l'une ou l'autre solution, le programmeur ne manquera de toutes façons pas de travail : trouver des solutions moins coûteuses à des problèmes actuels produira toujours de nouvelles opportunités un peu plus tard.

En supposant que le problème à résoudre ait été bien compris et qu'il faille écrire un nouveau programme, vous êtes prêt à commencer votre conception.

Le processus d'appréhension totale du problème (*analyse*) et d'élaboration d'un plan pour une solution (*conception*) est indispensable à l'obtention d'une application professionnelle de carrure internationale.

Votre environnement de développement

Pour utiliser ce livre, nous supposons que vous disposez d'un compilateur permettant de saisir des données directement sur une "console" (par exemple une fenêtre de commande MS-DOS ou une fenêtre shell), c'est-à-dire sans vous préoccuper d'un environnement

graphique comme Windows ou Macintosh. Recherchez une option comme `console` ou `easy window` ou consultez la documentation de votre compilateur.

Votre compilateur peut faire partie d'un environnement de développement intégré (IDE) ou posséder son propre éditeur de texte pour saisir le code source des programmes. Vous pouvez également utiliser un éditeur de texte séparé ou un logiciel de traitement de texte du moment que vous produisez des fichiers texte sans formatage ni style particuliers. Le Bloc-notes de Windows, l'éditeur Edit de DOS, les éditeurs Brief, Epsilon, Emacs et vi sont particulièrement bien adaptés à la saisie des codes sources. Si vous disposez d'un traitement de texte tel que WordPerfect ou Word (ou autre), utilisez la commande d'enregistrement au format texte simple.

Les documents créés à partir d'un éditeur de texte s'appellent des fichiers sources. En C++, ils portent traditionnellement l'extension `.cpp`, `.cp` ou `.c`. Dans cet ouvrage, nous avons choisi l'extension `.cpp`. Vérifiez que votre compilateur la prend en charge.



La plupart des compilateurs C++ acceptent toutes les extensions et affectent par défaut l'extension `.cpp` aux fichiers sources. Toutefois, soyez prudents, car certains compilateurs traitent les fichiers `.c` comme du code C et les fichiers `.cpp` comme du code C++. Vérifiez la documentation du compilateur. Dans tous les cas, une utilisation cohérente des `.cpp` pour les fichiers du code source C++ facilitera la tâche des programmeurs qui devront comprendre votre code.

Faire	*Ne pas faire*
● Écrire le fichier source à l'aide de l'éditeur de texte fourni avec le compilateur ou d'un éditeur externe. ● Enregistrer votre fichier source avec l'extension `.c`, `.cp` ou `.cpp`. ● Consulter la documentation du compilateur et de l'éditeur de liens pour connaître les différentes étapes de la création du programme.	● Utiliser les fonctions de mise en forme d'un traitement de texte. Si vous utilisez un traitement de texte, sauvegardez le fichier source au format texte ASCII. ● Utiliser une extension `.c` si votre compilateur considère ces fichiers comme du code C et non du code C++.

Création du programme

La première étape de la création d'un programme consiste à écrire les commandes adaptées (instructions) dans un fichier source. Même si les instructions du fichier source semblent quelque peu mystérieuses pour ceux qui ne connaissent pas C++, il s'agit quand même d'un format lisible. Le fichier source n'est pas un programme : il vous sera impossible de le lancer ou de l'exécuter comme vous le feriez avec un programme exécutable.

Création d'un fichier objet à l'aide du compilateur

Pour transformer le code source en programme, vous allez utiliser un compilateur. L'invocation et la configuration de cet outil variant d'un compilateur à un autre, il est vivement recommandé de consulter la documentation.

La compilation produit un fichier objet. Le plus souvent, il porte l'une des extensions `.obj` ou `.o`. Toutefois, il ne s'agit pas encore d'un programme exécutable. Pour créer le fichier `.exe`, vous devez utiliser l'éditeur de liens (*linker*).

Création d'un fichier exécutable à l'aide de l'éditeur de liens

Les programmes C++ sont produits en liant un ou plusieurs fichiers objet (`.obj` ou `.o`) avec une ou plusieurs bibliothèques. Une bibliothèque regroupe des fichiers fournis avec le compilateur, achetés séparément ou que vous créez et compilez vous-même. Tous les compilateurs C++ sont livrés avec une bibliothèque de fonctions et de classes très utiles, que vous pouvez inclure dans vos programmes. Nous traiterons les classes et les fonctions plus en détail ultérieurement.

La création d'un fichier exécutable se décompose en trois étapes :

1. Création d'un fichier source portant l'extension `.cpp`.
2. Compilation du code source en un fichier objet portant l'une des extensions `.obj` ou `.o`.
3. Liaison du fichier objet avec les différentes bibliothèques, afin d'obtenir le programme exécutable.

Cycle de développement

Si chaque programme fonctionnait parfaitement dès le premier coup, le cycle de développement complet serait : écriture du programme, compilation du code source, édition des liens et lancement de l'application. Malheureusement, rares sont les programmes qui ne contiennent aucune erreur. Certaines feront échouer la compilation du programme, d'autres l'édition des liens, d'autres encore ne se déclareront qu'au moment de l'exécution (celles-ci sont souvent appelées "bogues").

Toute erreur, quelle qu'elle soit doit être corrigée. Ceci implique de modifier le code source, de le compiler à nouveau, de recréer les liens et de relancer le programme. La Figure 1.1 illustre les différentes étapes de ce cycle de développement.

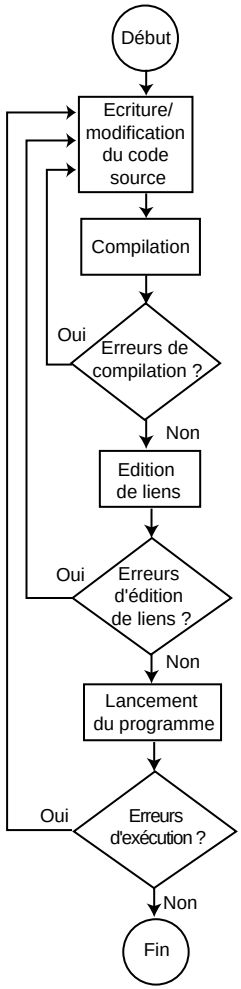


Figure 1.1
Les étapes du cycle de développement d'un programme C++.

Votre premier programme C++ : *BONJOUR.cpp*

Les ouvrages traitant de programmation proposent traditionnellement un programme qui permet d'afficher "Bonjour" à l'écran. Nous ne dérogerons donc pas à cette règle !

Ouvrez l'éditeur de texte, puis tapez le code source du Listing 1.1 en respectant scrupuleusement la syntaxe (ne tenez pas compte de la numérotation des lignes qui ne sert ici qu'à clarifier les choses). Après vérification, enregistrez le fichier, compilez-le, éditez les liens, puis lancez-le. Si tout se passe correctement, "Bonjour !" apparaît à l'écran. Ne vous souciez pas des instructions utilisées ici ; pour le moment, nous nous contentons de vous présenter

le cycle de développement. En fait, les différents aspects de ce programme seront traités dans les deux prochains chapitres.



Le listing suivant est numéroté à gauche afin de vous permettre de suivre pas à pas son déroulement. Vous ne devez pas taper ces numéros. Dans le premier exemple, vous devez uniquement taper pour la première ligne :

```
#include <iostream>
```

Listing 1.1 : Programme *bonjour.cpp*

```
1:  #include <iostream>
2:
3:  int main()
4:  {
5:    std::cout << "Bonjour !\n";
6:    return 0;
7:  }
```

Vérifiez que ce que vous avez tapé est conforme à ce fichier source. Attention à la ponctuation ! Pour obtenir le symbole de redirection (<<) à la ligne 5, appuyez deux fois sur la touche <. Les termes `std` et `cout` sont séparés par deux caractères deux-points (:). N'oubliez pas non plus le point-virgule qui termine les lignes 5 et 6.

La plupart des compilateurs compilent les fichiers et réalisent l'édition de liens dans la foulée. Reportez-vous à votre documentation pour voir si vous devez fournir une option particulière ou exécuter une commande pour effectuer cette édition des liens. En cas d'erreur, comparez votre fichier source avec celui du Listing 1.1. Si le compilateur signale que le fichier `iostream` est introuvable, vous devrez peut-être consulter votre documentation pour savoir comment votre compilateur préconise de configurer les chemins d'accès des fichiers à inclure et les variables d'environnement.

Si un message vous informe qu'il n'existe pas de prototype pour la fonction `main`, tapez l'instruction `int main() ;` avant la ligne 3 (c'est l'une de ces vilaines variations entre compilateurs). Si tel est le cas, vous devrez inclure cette instruction dans tous les programmes de cet ouvrage, bien que la plupart des compilateurs n'en auront pas besoin.

Votre programme, dans ce cas, ressemblera à ceci :

```
#include <iostream>
int main(); // ligne inutile pour la plupart des compilateurs
int main()
{
    std::cout <<"Bonjour !\n";
    return 0;
}
```

Sous Windows, essayez de lancer `bonjour.exe` (en remplaçant l'extension de l'exécutable par celle de votre système d'exploitation ; sous Unix, par exemple, exécutez `bonjour`, car les exécutables ne possédent pas d'extension). L'exécution du programme doit afficher :

```
Bonjour !
```

Si c'est le cas, bravo ! Tout va pour le mieux. Vous venez de créer votre premier programme C++. Il n'est peut-être pas spectaculaire, mais c'est ainsi qu'ont débuté la plupart des programmeurs professionnels.

Certains programmeurs utilisant des IDE (comme Visual Studio ou Borland C++ Builder) risquent de voir apparaître brièvement le programme dans une fenêtre qui disparaîtra rapidement, sans laisser le temps de lire ce qu'elle contient. Dans ce cas, ajoutez ces lignes à votre source, juste avant l'instruction `return` :

```
char reponse;
std::cin >> reponse;
```

Ces lignes forcent le programme à faire une pause jusqu'à ce que vous ayez saisi un caractère suivi d'un appui sur la touche Entrée. Elles permettent donc de voir les résultats de votre test. Si elles sont nécessaires pour le fichier `bonjour.cpp`, elles le seront sûrement pour la plupart des programmes de cet ouvrage.

Utilisation des bibliothèques standard

Si votre compilateur est très ancien, il peut refuser de compiler le code ci-dessus car il ne trouvera pas les nouveaux fichier entêtes du standard ANSI. Dans ce cas, modifiez votre programme de la façon suivante :

```
1:  #include <iostream.h>
2:
3:  int main()
4:  {
5:        cout << "Bonjour !\n";
6:        return 0;
7:  }
```

Vous remarquerez que le nom du fichier entête se termine à présent par `.h` (point-h) et que nous n'utilisons plus `std::` devant `cout` à la ligne 5. Ceci correspond à l'ancien style (pré-ANSI) de fichiers entête et signifie que votre compilateur est trop ancien ; il conviendra pour les premiers chapitres, mais lorsque nous aborderons les modèles et les exceptions, il risque d'être inutilisable.

Comment tirer parti de votre compilateur

Ce livre n'est *pas* spécifique à un compilateur donné ; les programmes inclus fonctionneront donc avec *n'importe quel* compilateur C++ compatible ANSI, sur n'importe quelle plate-forme (Windows, Macintosh, Unix, Linux, etc.).

Ceci dit, une grande majorité de programmeurs travaillent sous l'environnement Windows et la plupart des développeurs professionnels ont recours aux compilateurs Microsoft. Il n'est pas possible d'indiquer ici les détails de la compilation et de l'édition de liens avec tous les compilateurs possibles, mais nous pouvons vous montrer comment débuter avec Microsoft Visual C++ ; cela ne devrait pas être très différent avec votre compilateur, bien qu'il soit conseillé de vous reporter à sa documentation spécifique.

Construction du projet Bonjour

Les étapes nécessaires à la création et au test du programme Bonjour sont les suivantes :

1. Lancez le compilateur.
2. Choisissez Nouveau/Projet dans le menu Fichier.
3. Choisissez Application Console Win32 et entrez un nom pour le projet, *Bonjour*, par exemple, puis cliquez sur Terminer.
4. Sélectionnez Projet vide dans les choix proposés et cliquez sur OK. Une boîte de dialogue contenant des informations sur le nouveau projet s'affiche.
5. Cliquez sur OK. Vous revenez à la fenêtre principale de l'éditeur.
6. Choisissez Nouveau dans le menu Fichier.
7. Choisissez Fichier C++ et donnez-lui un nom : *bonjour*, par exemple, que vous saisirez dans la zone de texte Nom du fichier.
8. Cliquez sur OK. Vous revenez à la fenêtre principale de l'éditeur.
9. Saisissez le code du Listing précédent.
10. Choisissez Générer bonjour dans le menu Générer.
11. Vérifiez que vous n'avez pas d'erreurs de compilation. Vous trouverez ces informations au bas de l'éditeur.
12. Exécutez le programme en appuyant sur Ctrl+F5.
13. Appuyez sur la barre d'espacement pour terminer le programme.

Le programme s'exécute, mais l'affichage est si rapide que je n'ai pas le temps de lire. Qu'est-ce qui ne va pas ?

Consultez la documentation de votre compilateur ; vous devez pouvoir contraindre le programme à respecter une pause après son exécution. Avec les compilateurs Microsoft il suffit d'utiliser la combinaison de touches Ctrl+F5.

Avec n'importe quel compilateur, vous pouvez insérer les lignes suivantes immédiatement avant l'instruction `return` (c'est-à-dire entre les lignes 5 et 6 du Listing 1.1) :

```
char reponse;
std::cin >> reponse;
```

Le programme fera une pause pour attendre que vous entriez une valeur. Pour terminer le programme, entrez un nombre (par exemple 1) puis appuyez sur Entrée (si nécessaire).

La signification des termes `std::cin` et `std::cout` sera présentée dans les prochains chapitres ; pour l'instant, utilisez-les sans vous poser de questions.

Erreurs de compilation

Les erreurs de compilation peuvent survenir dans un certain nombre de cas. Le plus souvent, il s'agit d'une faute de frappe ou d'un oubli. Les bons compilateurs mettront la ligne incriminée en évidence et vous indiqueront l'erreur que vous avez commise. Les plus performants proposeront même une solution au problème rencontré !

Pour tester les fonctionnalités de votre compilateur, il suffit d'insérer délibérément une erreur dans le programme *bonjour.cpp*, en ôtant la ligne 7 où doit se trouver l'accolade fermante.

Listing 1.2 : Erreur de compilation

```
1:  #include <iostream>
2:
3:  int main()
4:  {
5:    std::cout << "Bonjour !\n";
6:    return 0;
```

Compilez de nouveau le programme. Votre compilateur affiche une erreur similaire à celle-ci :

```
bonjour.cpp, line 7 : fatal error C1004: unexpected end of file found.
```

Vous constatez que le message est un petit peu énigmatique et qu'il y est question d'une erreur qui correspond à la ligne 7. Ici, le compilateur indique qu'il manque des lignes sources et qu'il a atteint la fin du fichier sans détecter l'accolade fermante. En règle générale, les messages d'erreur contiennent un numéro de ligne, ce qui permet de résoudre le problème plus aisément.

Parfois, ces messages ne vous indiqueront que la proximité du problème : si un compilateur pouvait parfaitement identifier toutes les erreurs, il les corrigerait lui-même.

Questions-réponses

Q Quelle est la différence entre un éditeur de texte et un traitement de texte ?

R Contrairement à un traitement de texte, un éditeur de texte génère des fichiers texte sans code de formatage ni de mise en page. Un simple éditeur de texte contient du texte brut, sans caractères gras, italiques, etc.

Q Mon compilateur comprend un éditeur intégré. Dois-je l'utiliser ?

R La plupart des compilateurs seront capables de traiter du code tapé à l'aide d'un éditeur de texte. Un éditeur intégré permet de se déplacer rapidement dans le fichier source et de suivre pas à pas le cycle de développement avant et après compilation. Les compilateurs modernes présentent l'avantage de pouvoir consulter l'aide en ligne, de compiler le code en direct et de résoudre les problèmes sans quitter l'environnement de développement.

Q Le compilateur a produit des messages d'avertissement. Puis-je les ignorer ?

R Les compilateurs produisent généralement des avertissements et des erreurs. En cas d'erreurs, le programme ne pourra pas être totalement construit. En ce qui concerne les avertissements, le compilateur poursuivra son travail et créera tout de même le programme.

Bien que de nombreux livres ne donnent pas de réponse à cette question, la nôtre est *non* ! Dès maintenant, considérez les messages d'avertissement comme des messages d'erreur. Le compilateur vous informe que quelque chose d'anormal ou de suspect a été détecté. Lisez le message à l'écran, puis corrigez l'erreur. Certains compilateurs disposent même d'un paramètre qui permet de traiter tous les avertissements comme des erreurs et donc d'empêcher que le programme ne devienne un exécutable.

Q Qu'est-ce que la compilation ?

R La véritable compilation intervient lorsque vous lancez le compilateur. Il s'agit de l'étape qui précède l'édition de liens et le lancement du programme. Les programmeurs ont

tendance à utiliser ce terme comme un raccourci désignant la véritable étape de compilation suivie de l'édition de liens.

Testez vos connaissances

1. Quelle est la différence entre un interpréteur et un compilateur ?
2. Comment compiler un fichier source ?
3. À quoi sert l'éditeur de liens ?
4. Quelles sont les différentes étapes du cycle de développement ?

Exercices

1. Examinez le listing qui suit et, sans l'exécuter, devinez ce qu'il permet de faire.

```
 1: #include <iostream>
 2: int main()
 3: {
 4:   int x = 5;
 5:   int y = 7;
 6:   std::cout << end1;
 7:   std::cout << x + y << " " << x * y;
 8:   std::cout << end;
 9:   return 0;
10: }
```

2. Tapez le programme de l'exercice précédent. Compilez-le et éditez les liens. Que fait-il ? Vous étiez-vous trompé en répondant à la question précédente ?
3. Tapez le programme suivant, puis compilez-le. Quelle erreur produit-il ?

```
1: include <iostream>
2: int main()
3: {
4:   std::cout << "Salut \n";
5:   return 0;
6: }
```



2

Anatomie d'un programme C++

Au sommaire de ce chapitre

- La structure d'un programme C++
- Les interactions entre ses composants
- La nature et le rôle d'une fonction

Un programme C++ se compose d'objets, de fonctions, de variables et d'un certain nombre d'autres composants. L'essentiel de cet ouvrage est consacré à l'explication détaillée de ces différents éléments mais, pour comprendre comment ils s'articulent ensemble, il convient d'étudier un programme complet.

Un programme simple

Le programme *bonjour.cpp* du Chapitre 1 est intéressant à bien des égards. Pour information, le Listing 2.1 reproduit la version originale du fichier source.

Listing 2.1 : Le programme *bonjour.cpp* présente une structure type d'application C++

```
1:  #include <iostream>
2:
3:  int main()
4:  {
5:      std::cout << "Bonjour !\n";
6:      return 0;
7:  }
```

Ce programme affiche le résultat suivant :

```
Bonjour !
```

La ligne 1 inclut le fichier `iostream` dans le fichier courant.

Le principe est le suivant : le premier caractère est le signe dièse (`#`) qui indique une directive adressée au préprocesseur. Chaque fois que vous lancez le compilateur, le préprocesseur s'exécute en premier : son rôle est de détecter les lignes commençant par le signe dièse et d'effectuer certaines opérations en fonction de ce préfixe. Le préprocesseur sera étudié en détail au Chapitre 21.

La commande `#include` est une instruction du préprocesseur qui signifie : "Le terme qui suit est un nom de fichier. Trouver ce fichier et insérer ici son contenu". Les chevrons (< et >) placés autour du nom du fichier demandent au préprocesseur de rechercher ce fichier aux emplacements habituels. Si l'installation a été réalisée correctement, le préprocesseur trouvera le fichier `iostream` dans le répertoire contenant tous les fichiers `include`. Ce fichier (dont le nom est l'abréviation de *Input-Output-Stream*) est utilisé par `cout`, qui permet d'afficher le résultat à l'écran. Le rôle de la ligne 1 est d'inclure le fichier `iostream` dans ce programme, comme si vous l'aviez vous-même saisi.



Le préprocesseur est lancé avant le compilateur. Il traduit toutes les lignes commençant par le signe dièse (`#`) en commandes spéciales qui rendent votre code directement exploitable par le compilateur.



Les compilateurs ne sont pas tous cohérents dans leur prise en charge de `#include` lorsque l'on omet l'extension des fichiers. Si vous obtenez des messages d'erreur, vous devrez peut-être modifier le chemin de recherche de `include` pour votre compilateur ou ajouter l'extension au fichier inclus.

Le programme commence vraiment en ligne 3, avec la fonction `main()`. Cette fonction est obligatoire dans tous les programmes C++. Une fonction est un bloc d'instructions qui effectue une ou plusieurs actions et qui peut être appelée par une autre fonction, mais `main()` est spéciale car elle est appelée automatiquement au lancement du programme.

Comme toutes les fonctions, `main()` doit indiquer le type de la valeur qu'elle renvoie. Dans le programme *bonjour.cpp*, il s'agit de `int`, ce qui signifie que la fonction renverra une valeur entière au système d'exploitation lorsqu'elle se terminera. Ici, elle renvoie la valeur 0, comme le montre la ligne 6. Cette valeur renvoyée a relativement peu d'importance et est assez rarement utilisée, mais le standard C++ exige que la fonction `main()` soit déclarée ainsi.



Certains compilateurs permettent de déclarer `main()` comme renvoyant `void`, ce qui signifie qu'aucune valeur n'est renvoyée. Ceci n'est plus autorisé en C++ et il est préférable de ne pas prendre cette mauvaise habitude. Déclarez `main()` comme renvoyant `int` et renvoyez simplement la valeur 0 à la dernière ligne de la fonction `main()`.



Tous les systèmes d'exploitation permettent de tester la valeur renvoyée par un programme. La convention consiste à renvoyer 0 pour indiquer que le programme s'est terminé correctement, et une valeur différente de zéro dans le cas contraire.

Toutes les fonctions commencent par une accolade ouvrante (`{`) et se terminent par une accolade fermante (`}`). Dans notre programme, les accolades de la fonction `main()` figurent aux lignes 4 et 7. Toute instruction comprise entre ces deux accolades fait partie de la fonction.

Le plat de résistance du programme se trouve à la ligne 5.

L'objet `cout` permet d'afficher un message à l'écran. Les objets et leurs particularités seront présentés au Chapitre 6 et l'objet `cout` et son alter ego `cin` seront décrits au Chapitre 17. En C++, ces deux objets permettent de gérer respectivement les sorties (notamment à l'écran) et les entrées (au clavier, par exemple).

`cout` est un objet fourni par la bibliothèque standard. Une bibliothèque est une collection de classes. La bibliothèque standard est fournie avec tout compilateur compatible ANSI.

On indique au compilateur que l'objet `cout` que l'on souhaite utiliser fait partie de la bibliothèque standard à l'aide du spécificateur d'espace de nom `std`. Vous pouvez en effet avoir des objets portant le même nom, mais provenant de fournisseurs différents et c'est la raison pour laquelle C++ prévoit l'utilisation des "espaces de noms". Un espace de nom est une façon de dire : "Lorsque je dis `std::cout`, je fais référence à l'objet `cout` faisant

partie de l'espace de nom standard, et d'aucun autre". Vous le faites savoir au compilateur en indiquant le terme `std` suivi de deux caractères deux-points avant l'objet `cout`. Nous reparlerons des espaces de noms ultérieurement.

La syntaxe de `cout` est relativement simple. Ce terme est suivi de l'opérateur de redirection de sortie (<<). La variable ou la chaîne de caractères figurant immédiatement après << sera écrite à l'écran. Si vous voulez produire un message directement à l'écran, n'oubliez pas de le mettre entre guillemets ("), comme à la ligne 5.



L'opérateur de redirection est représenté par deux signes "plus grand que" sans espace entre eux.

Une *chaîne de caractères* est une suite de caractères imprimables.

Les deux derniers caractères, `\n`, correspondent à un saut de ligne qui sera produit après le message. Pour en savoir plus sur les retours chariot et les sauts de ligne, reportez-vous au Chapitre 18.

La fonction `main()` se termine à la ligne 7 avec l'accolade fermante.

Étude rapide de l'objet *cout*

Au Chapitre 17, vous apprendrez à afficher des données à l'écran à l'aide de l'objet `cout`. Pour le moment, vous pouvez l'utiliser sans connaître parfaitement son mode de fonctionnement. Pour afficher une valeur sur votre moniteur, tapez le mot `cout`, suivi de l'opérateur d'insertion (<<), c'est-à-dire de deux signes "inférieur à". Même si le signe est constitué de deux caractères, C++ considère qu'il n'y en a qu'un.

Tapez les données après cet opérateur. Le Listing 2.2 est un exemple d'utilisation de l'opérateur d'insertion. Recopiez le fichier source de ce petit programme en indiquant vos propres prénom et nom.

Listing 2.2 : Exemple d'utilisation de *cout*

```
1:  // Listing 2.2 utilisation de std::cout
2:  #include <iostream>
3:  int main()
4:  {
5:      std::cout << "Salut.\n";
6:      std::cout << "Je tape 5 : " << 5 << "\n";
7:      std::cout << "L'opérateur std::endl ";
8:      std::cout << "provoque un saut de ligne à l'écran.";
```

```
 9:      std::cout <<  std::endl;
10:      std::cout << "Voici un tres grand nombre :\t" << 70000;
11:      std::cout << std::endl;
12:      std::cout << "8 et 5 font :\t";
13:      std::cout << 8+5 << std::endl;
14:      std::cout << "Voici une fraction : \t\t";
15:      std::cout << (float) 5/8 << std::endl;
16:      std::cout << "Et un nombre astronomique : \t";
17:      std::cout << (double) 7000 * 7000 << std::endl;
18:      std::cout << "Remplacez le nom ";
19:      std::cout << "par le votre...\n";
20:      std::cout << "Elie Kopter est un programmeur C++ ! \n";
21:      return 0;
22:  }
```

Ce programme produit le résultat suivant :

```
Salut.
Je tape 5 : 5
L'opérateur std::endl provoque un saut de ligne à l'écran.
Voici un très grand nombre :     70000
8 et 5 font :                    13
Voici une fraction :             0.625
Et un nombre astronomique :      4.9e+07
Remplacez le nom par le votre ...
Elie Kopter est un programmeur C++ !
```



Certains compilateurs ont un bogue qui exige que des parenthèses soit placées autour de l'addition avant de la passer à `cout`*. La ligne 13 devient alors :*

```
13: cout << (8+5) << std::endl;
```

À la ligne 2, l'instruction `#include <iostream>` intègre le fichier `iostream` au code source. Celui-ci est nécessaire à `cout` et à ses fonctions associées.

À la ligne 5, `cout` est utilisé dans sa version la plus simple pour afficher une chaîne de caractères. Le symbole `\n` est un caractère de formatage spécial qui demande à `cout` d'afficher un caractère "nouvelle ligne".

Trois valeurs sont ensuite passées à `cout` à la ligne 6. Chacune d'entre elles est séparée par l'opérateur d'insertion. La première valeur correspond à `"Je tape 5 : "`. Notez que l'espace situé après le caractère deux-points fait partie de la chaîne. Le chiffre 5 est alors passé à l'opérateur d'insertion, puis le programme saute une ligne (entre guillemets ou apostrophes). La ligne suivante apparaît à l'écran :

```
Je tape 5 : 5
```

Le chiffre suit immédiatement la chaîne de caractères, car vous n'avez pas inséré de saut de ligne. Sans le savoir, vous avez *concaténé* deux valeurs.

La ligne 7 affiche un message puis utilise le manipulateur `std::endl` qui a pour but d'écrire une nouvelle ligne à l'écran (Pour en savoir plus sur `endl`, reportez-vous au Chapitre 16). `endl` faisant aussi partie de la bibliothèque standard, on utilise le préfixe `std::`, comme pour `cout`.



`endl` signifie end line (ligne finale). La dernière lettre est bien la lettre "l" et non le chiffre 1.

L'utilisation de `endl` est préférable à celle de `\n`, car il est adapté au système d'exploitation utilisé. En revanche, `\n` risque de ne pas représenter le caractère de nouvelle ligne complet exigé sur un système d'exploitation ou une plate-forme particuliers.

À la ligne suivante, vous découvrez un nouveau caractère de formatage : `\t`. Il permet d'insérer une tabulation pour aligner les chaînes de caractères (lignes 10 à 16). La ligne 10 montre que l'on peut afficher tous les nombres entiers, quel que soit leur type. À la ligne 14, on passe une addition (8 + 5) à `cout`, qui affiche 13.

À la ligne 15, la fraction 5/8 est transmise à `cout`. Le terme (`float`) indique qu'une valeur décimale (à virgule flottante) doit s'afficher. L'expression `7000 * 7000` est ensuite transmise à `cout` (ligne 17) et le terme `double` indique à `cout` qu'il s'agit d'une valeur à virgule flottante. Pour obtenir des informations détaillées sur les types de données, reportez-vous au Chapitre 3.

Aux lignes 18 et 20, vous avez remplacé le nom du programmeur par le vôtre. Si c'est le cas, la sortie confirmera que vous êtes un programmeur C++, ce qui doit être vrai puisque c'est l'ordinateur qui le dit !

Utilisation de l'espace de nom standard

L'utilisation systématique de `std::` devant `cout` et `endl` pouvant devenir plutôt fastidieuse à la longue, la norme ANSI propose deux solutions à ce petit problème.

La première consiste à indiquer au compilateur, au début du code, que vous allez utiliser les objets `cout` et `endl` de la bibliothèque standard, comme au Listing 2.3 (lignes 5 et 6).

Listing 2.3 : Utilisation du mot-clé *using*

```
1:  // Listing 2.3 - Utilisation du mot-clé using
2:  #include <iostream>
```

```
 3:  int main()
 4:  {
 5:      using std::cout;
 6:      using std::endl;
 7:
 8:      cout << "Salut.\n";
 9:      cout << "Je tape 5 : " << 5 << "\n";
10:      cout << "L'opérateur endl ";
11:      cout << "provoque un saut de ligne a l'écran.";
12:      cout <<  endl;
13:      cout << "Voici un très grand nombre :\t" << 70000;
14:      cout <<  endl;
15:      cout << "8 et 5 font :\t";
16:      cout << 8+5 << endl;
17:      cout << "Voici une fraction:\t\t";
18:      cout << (float) 5/8 << endl;
19:      cout << "Et un nombre astronomique :\t";
20:      cout << (double) 7000 * 7000 << endl;
21:      cout << "Remplacez le nom ";
22:      cout << "par le votre...\n";
23:      cout << "Elie Kopter est un programmeur C++ !\n";
24:      return 0;
25:  }
```

Ce qui produit le résultat suivant :

```
Salut.
Je tape 5 : 5
L'opérateur endl provoque un saut de ligne à l'écran.
Voici un tres grand nombre :     70000
8 et 5 font :                    13
Voici une fraction :             0.625
Et un nombre astronomique :      4.9e+07
Remplacez le nom par le vôtre ...
Elie Kopter est un programmeur C++ !
```

La sortie est identique. Les lignes 5 et 6 du Listing 2.3 informent le compilateur à l'aide du mot-clé `using`, que nous allons utiliser deux objets de la bibliothèque standard. Il n'est dans ce cas plus nécessaire de qualifier les objets `cout` et `endl`.

Une autre méthode consiste à informer le compilateur que nous allons utiliser l'espace de nom standard en totalité ; c'est-à-dire que tout objet non explicitement qualifié sera considéré comme provenant de l'espace de nom standard. Au lieu d'écrire `std::cout;` nous allons simplement préciser, comme au Listing 2.4 : `using namespace std;`.

Listing 2.4 : Utilisation du mot-clé *namespace*

```
 1:  // Listing 2.4 - Utilisation de l'espace de nom standard
 2:  #include <iostream>
 3:  int main()
 4:  {
 5:      using namespace std;
 6:
 7:      cout << "Salut.\n";
 8:      cout << "Je tape 5 : " << 5 << "\n";
 9:      cout << "L'opérateur endl ";
10:      cout << "provoque un saut de ligne à l'écran.";
11:      cout <<  endl;
12:      cout << "Voici un très grand nombre :\t" << 70000;
13:      cout <<  endl;
14:      cout << "8 et 5 font :\t";
15:      cout << 8+5 << endl;
16:      cout << "Voici une fraction:\t\t";
17:      cout << (float) 5/8 << endl;
18:      cout << "Et un nombre astronomique :\t";
19:      cout << (double) 7000 * 7000 << endl;
20:      cout << "Remplacez le nom indique";
21:      cout << "par le votre...\n";
22:      cout << "Elie Kopter est un programmeur C++ !\n";
23:      return 0;
24:  }
```

La sortie sera ici encore identique à celle des précédentes versions de ce programme. L'intérêt d'écrire `using namespace std;` est qu'il n'est plus nécessaire de désigner nommément les objets que vous allez utiliser (`cout` et `endl`). Vous prenez par contre le risque d'utiliser par inadvertance des objets d'une bibliothèque inadéquate.

Les puristes préfèrent écrire `std::` devant chaque instance de `cout` et `endl`. Les paresseux préféreront `using namespace std;`.

Commentaires

Lors de la saisie d'un programme, vous savez avec précision ce que vous faites. Mais si vous rouvrez le fichier source un mois plus tard, certains passages du listing risquent de vous paraître difficiles à comprendre.

Pour éviter ce désagrément et permettre aux autres de comprendre vos fichiers, n'hésitez pas à insérer des commentaires. Les commentaires sont du texte simple qui n'est pas traité

par le compilateur. Ils ont pour but d'informer le lecteur sur telle ou telle action du programme.

Les différents types de commentaires

C++ utilise deux types de commentaires : les commentaires sur une seule ligne et les commentaires multilignes.

Les premiers sont introduits par deux caractères slashs (`//`) qui indiquent au compilateur qu'il doit ignorer tout ce qui suit jusqu'à la fin de la ligne.

Les seconds sont introduits par les caractères slash-étoile (`/*`) et demandent au compilateur de ne pas traiter les caractères qui suivent, jusqu'au symbole de fin de commentaires étoile-slash (`*/`). Ces marques peuvent se trouver sur la même ligne ou être séparées par plusieurs lignes. À chaque symbole `/*` doit correspondre un symbole de fin de commentaires `*/`.

En général, les programmeurs C++ utilisent le double slash, puis une ligne de commentaire. Ils réservent les commentaires de plusieurs lignes pour délimiter de grandes parties d'un programme. Vous pouvez inclure des commentaires d'une ligne dans un bloc "commenté" : ce qui figure entre les symboles de commentaires multilignes n'est pas pris en compte, y compris les doubles slashs.



Les commentaires sur plusieurs lignes sont connus sous le nom de "style C" car ils ont été introduits par le langage C. Les commentaires d'une ligne sont apparus avec le C++ et sont donc devenus le "style C++". Les normes actuelles de ces deux langages intègrent les deux styles.

Utilisation des commentaires

Certains préconisent d'écrire des commentaires au début de chaque fonction pour indiquer son rôle et les valeurs qu'elle renvoie.

Cette recommandation est discutable, car ce type de commentaire mis en en-tête n'est pratiquement jamais remis à jour lors des différentes évolutions du programme. Les noms de fonctions doivent être suffisamment explicites pour éviter toute ambiguïté et les portions de code trop énigmatiques doivent être réécrites pour être suffisamment claires par elles-mêmes. Les commentaires ne doivent pas être une excuse d'un code obscur.

Ceci ne veut pas dire que les commentaires ne servent à rien, mais simplement qu'ils ne doivent pas se substituer à un code lisible : il est toujours préférable de clarifier un code plutôt que d'expliquer ce qu'il fait. Bref, écrivez du code lisible et n'utilisez les commentaires que pour fournir des informations supplémentaires.

Le Listing 2.5 va vous familiariser avec l'usage des commentaires. Il démontre que ceux-ci n'influent en rien sur le traitement et sur l'affichage des données.

Listing 2.5 : *aide.cpp* montre l'usage des commentaires

```
 1:  #include <iostream>
 2:
 3:  int main()
 4:  {
 5:      using std::cout;
 6:
 7:      /* Ceci est un commentaire qui se termine
 8:      lorsque le symbole étoile-slash
 9:      est rencontré */
10:      cout << "Bonjour!\n";
11:      // Ce commentaire se termine à la fin de la ligne
12:      cout << "Exemples de commentaires !\n";
13:
14:      // Les commentaires double slash peuvent occuper une seule
15:      /* ligne tout comme les commentaires de type slash-étoile */
16:      return 0;
17:  }
```

Ce programme produit le résultat suivant :

```
Bonjour !
Exemples de commentaires !
```

Les commentaires des lignes 7 à 9 sont ignorés par le compilateur, tout comme ceux des lignes 11, 14 et 15. Les commentaires de la ligne 11 sont compris entre le symbole `//` et le retour à la ligne, alors que ceux des lignes 7 et 15 requièrent un symbole de fermeture.



Certains compilateurs C++ reconnaissent un troisième type de commentaires : les commentaires de documentation, signalés par trois slashes (`///`). Les compilateurs qui les acceptent permettent de produire de la documentation à partir de ces commentaires. Cependant, cette syntaxe ne fait pas encore partie de la norme C++ et ne sera donc pas détaillée ici.

Quelques conseils...

Évitez d'insérer des commentaires lorsque l'opération effectuée est évidente. En effet, ils surchargent le fichier source et risquent d'être oubliés par le programmeur lors de la prochaine mise à jour. Cela étant dit, ce qui peut sembler évident à une personne peut paraître obscur à une autre : vous devez donc faire preuve de discernement lorsque vous placez des commentaires.

En résumé, les commentaires ne doivent pas dire ce qui se passe, mais expliquer *pourquoi* cela arrive.

Les fonctions

Bien que `main()` soit une fonction, ses caractéristiques la distinguent des autres. En effet, l'utilité des fonctions standard est qu'elles peuvent être *appelées* ou *invoquées* à tout moment dans un programme ; or la fonction `main()` est appelée par le système d'exploitation.

Le déroulement d'un programme est séquentiel (c'est-à-dire qu'il suit l'ordre des lignes de code). Quand il rencontre une instruction d'appel d'une fonction, il se déroute pour exécuter cette dernière. Lorsque la fonction se termine, le programme continue son exécution avec la ligne placée immédiatement après l'appel de la fonction.

Comparons l'appel d'une fonction à une action de la vie courante : alors que vous dessinez un paysage, la mine de votre crayon se casse. Vous cessez immédiatement de dessiner afin de tailler votre crayon. Puis vous reprenez votre activité artistique là où vous l'aviez interrompue. Il en est de même dans un programme. Lorsque celui-ci a besoin d'un service, il fait appel à une fonction, puis reprend son déroulement une fois que celle-ci est terminée. Le Listing 2.6 illustre ce concept.



Pour plus d'informations sur les fonctions, reportez-vous au Chapitre 5. Les types de données prises en charge seront présentées au Chapitre 3. Ce chapitre-ci donne uniquement un aperçu des fonctions, car on les retrouvera dans la plupart des programmes C++.

Listing 2.6 : Appel de fonction dans un programme

```
 1:  #include <iostream>
 2:
 3:  // Fonction DemoFonction
 4:  // affiche un message d'information
 5:  void DemoFonction()
 6:  {
 7:      std::cout << "Dans la fonction DemoFonction \n";
 8:  }
 9:
10:  // Fonction main - affiche un message,
11:  // appelle DemoFonction, puis affiche
12:  // un second message.
13:  int main()
14:  {
15:      std::cout << "Dans la fonction main\n" ;
```

```
16:      DemoFonction();
17:      std::cout << "Retour dans main\n";
18:      return 0;
19:  }
```

Ce programme produit le résultat suivant :

```
Dans la fonction main
Dans la fonction DemoFonction
Retour dans main
```

La fonction `DemoFonction()` est définie aux lignes 6 à 8. Lorsqu'elle est appelée, elle affiche un message, puis rend la main au programme.

Le véritable programme commence à la ligne 13. À la ligne 15, la fonction `main()` affiche un message indiquant que l'on est dans la fonction principale. À la ligne suivante, la fonction `DemoFonction()` est appelée, ce qui signifie que les instructions de cette dernière vont s'exécuter. Dans le cas présent, la fonction se résume à un message (ligne 7). À la ligne 8, `DemoFonction()` se termine. Le programme reprend son exécution à la ligne 17 et la fonction `main()` affiche son dernier message.

Utilisation des fonctions

L'intérêt d'une fonction est de transmettre au programme une valeur dont il a besoin. Par exemple, une fonction qui additionne deux entiers renverra la somme obtenue sous la forme d'une valeur entière. En revanche, une fonction qui affiche simplement un message ne transmet pas de valeur au programme et doit donc être déclarée comme renvoyant une valeur `void`.

Une fonction se compose d'un *en-tête* et d'un corps. L'en-tête se divise lui-même en plusieurs parties : le type de la valeur renvoyée, le nom de la fonction et les paramètres qu'elle attend. Ces derniers permettent de transmettre des valeurs à la fonction. Par exemple, si la fonction additionne deux nombres, ceux-ci devront lui être passés en paramètre. La ligne ci-dessous est un en-tête caractéristique d'une fonction `Somme` qui attend deux valeurs entières (`a` et `b`), puis renvoie une valeur entière :

```
int Somme(int a, int b)
```

Un paramètre formel est une déclaration du type de la valeur transmise à la fonction. La valeur réelle passée par la fonction appelante s'appelle le paramètre réel (certains programmeurs utilisent également le terme d'argument).

Le corps d'une fonction se compose d'une accolade ouvrante, de zéro ou plusieurs instructions et d'une accolade fermante. Les instructions constituent le travail réel de la fonction.

Une fonction peut renvoyer une valeur à l'aide de l'instruction `return`. Le type de cette valeur doit correspondre à celui qui a été déclaré dans l'en-tête de la fonction. En outre, l'exécution de l'instruction `return` provoque également la fin de l'exécution de la fonction. Une fonction qui se termine sans passer par une instruction `return` renvoie `void` (rien) lorsqu'elle atteint son accodade fermante. En outre, une fonction censée renvoyer une valeur mais se terminant sans instruction `return` peut provoquer un message d'avertissement ou une erreur avec certains compilateurs.

Dans le Listing 2.7, la fonction attend deux paramètres entiers et renvoie une valeur entière. Pour le moment, ne vous attardez pas sur la syntaxe et sur le traitement des valeurs entières (comme `int x`). Vous découvrirez ces sujets au Chapitre 3.

Listing 2.7 : Mise en œuvre d'une fonction simple

```
 1:  #include <iostream>
 2:  int Somme (int x, int y)
 3:  {
 4:      std::cout << "Somme() a recu " << x << " et " << y << "\n";
 5:      return (x + y);
 6:  }
 7:
 8:  int main()
 9:  {
10:      using std::cout;
11:      using std::cin;
12:
13:
14:      cout << "Vous etes dans main() !\n";
15:      int a, b, c;
16:      cout << "Entrez deux nombres : ";
17:      cin >> a;
18:      cin >> b;
19:      cout << "\nAppel de Somme()\n";
20:      c = Somme(a, b);
21:      cout << "\nRetour dans main().\n";
22:      cout << "c vaut maintenant " << c;
23:      cout << "\nFin du traitement...\n\n";
24:      return 0;
25:  }
```

Ce programme produit le résultat suivant :

```
Vous etes dans main() !
Entrez deux nombres : 3 5

Appel de Somme()
```

```
Somme() a recu 3 et 5

Retour dans main().
c vaut maintenant 8
Fin du traitement...
```

Définie à la ligne 2, la fonction `Somme()` reçoit deux paramètres entiers et renvoie une valeur entière. Le programme commence à la ligne 8 ; il affiche un message invitant l'utilisateur à entrer deux nombres (ligne 16), séparés par un espace. Ces nombres sont placés dans les variables `a` et `b` aux lignes 17 et 18. Pour valider, tapez sur Entrée. La fonction `main()` passe ces deux nombres en paramètres à la fonction `Somme()` (ligne 20).

Le programme se déroute dans la fonction `Somme()` qui commence à la ligne 2. Les valeurs de `a` et `b` sont reçues, respectivement, par les paramètres `x` et `y`. La fonction affiche ces valeurs puis les additionne ; le résultat obtenu est renvoyé à la ligne 5, puis la fonction rend la main à la fonction qui l'a appelée – la fonction `main()`, ici.

Dans les lignes 17 et 18, l'objet `cin` permet d'obtenir des valeurs pour les variables `a` et `b`, qui sont ensuite affichées à l'aide de l'objet `cout`. Nous traiterons en détail les variables et les différents aspects d'un programme dans les chapitres suivants.

Méthodes ou fonctions

Il faut noter ici que les différents langages de programmation et les différentes méthodologies peuvent désigner les fonctions par un autre nom. L'un des termes les plus habituels est celui de *méthode*. Une méthode est simplement une fonction qui fait partie d'une classe.

Questions-réponses

Q Quelle est la signification du terme `#include` ?

R Il s'agit d'une directive du préprocesseur qui s'exécute lorsque vous appelez le compilateur. Cette directive insère le fichier dont le nom est indiqué entre chevrons, comme si vous l'aviez saisi en totalité, dans le code source du programme.

Q Quelle est la différence entre les commentaires `//` et les commentaires de type `/*` ?

R Les commentaires double slash (`//`) (commentaires sur une ligne) prennent fin avec le retour à la ligne, alors que les commentaires slash-étoile (`/*`) (commentaires sur plusieurs lignes) se terminent avec un symbole de fermeture (`*/`). N'oubliez pas de fermer tous les commentaires multilignes afin d'éviter toute erreur de compilation.

Q Qu'est-ce qui distingue un bon commentaire d'un mauvais ?

R Un bon commentaire indique les actions qui vont être réalisées par une fonction, une instruction ou un bloc d'instructions. Un mauvais commentaire surcharge le code de formules redondantes et d'explications inutiles. Le code des programmes doit être écrit de façon à se comprendre aisément par lui-même. Une ligne bien écrite ne devrait pas avoir besoin d'un commentaire pour expliquer ce qu'elle fait, mais c'est à vous de distinguer ce qui mérite commentaire et ce qui est évident pour tous.

Testez vos connaissances

1. Quelle est la différence entre un compilateur et un préprocesseur ?
2. Quelles sont les particularités de la fonction `main()` ?
3. Quels sont les deux types de commentaires admis en C++ ? En quoi diffèrent-ils ?
4. Est-il possible d'imbriquer des commentaires ?
5. Un commentaire peut-il être plus long qu'une ligne ?

Exercices

1. Écrivez un programme qui affiche "J'adore C++" à l'écran.
2. Écrivez le plus petit programme pouvant être compilé, lié, puis lancé.
3. CHERCHEZ L'ERREUR : compilez ce programme. Pourquoi échoue-t-il ? Corrigez l'erreur.

```
1: #include <iostream>
2: main()
3: {
4:   std::cout << Y a-t-il un bogue dans la salle ?";
5: }
```

4. Après avoir corrigé ce bogue, recompilez le programme puis lancez-le.
5. Incluez dans le Listing 2.7 une fonction de soustraction, nommée `Difference()` et utilisez-la de la même manière que la fonction `Somme()`. Passez-lui les valeurs qui ont été passées à la fonction `Somme()`.



3

Variables et constantes

Au sommaire de ce chapitre

- Déclarer et définir des variables et des constantes
- Affecter des valeurs aux variables, puis gérer ces valeurs
- Afficher le contenu d'une variable à l'écran

Les programmes doivent mémoriser les données qu'ils utilisent ou les créer pour les réutiliser lors de l'exécution du programme. Pour cela, les variables et les constantes proposent plusieurs méthodes pour représenter, stocker et manipuler des informations.

Qu'est-ce qu'une variable ?

En C++, une *variable* est un emplacement destiné à recevoir des données. Cette zone se situe dans la mémoire vive de votre ordinateur et reçoit une valeur qui peut ensuite être relue.

Les variables sont utilisées pour un stockage temporaire. Si vous quittez un programme et arrêtez votre ordinateur, l'information stockée dans ces variables est perdue. Un stockage

permanent est un autre problème. Généralement, pour stocker des valeurs de façon persistante, vous devez les enregistrer soit dans une base de données, soit dans un fichier sur le disque. Cette dernière méthode est présentée au Chapitre 16.

Stockage des données en mémoire

La mémoire de votre ordinateur peut être comparée à un ensemble de cases alignées les unes à côté des autres. Ces emplacements sont numérotés séquentiellement par des nombres appelés *adresses mémoire*. Une variable peut occuper une ou plusieurs cases afin de stocker la valeur qu'elle contient.

Le nom de la variable (par exemple, `maVariable`) est une étiquette placée sur l'une des cases. Il identifie la variable et vous épargne de devoir connaître son adresse réelle en mémoire. La Figure 3.1 est une représentation symbolique de l'agencement de la mémoire. Comme vous pouvez le constater, `maVariable` commence à l'adresse mémoire 103. En fonction de sa taille, elle peut occuper une ou plusieurs adresses.

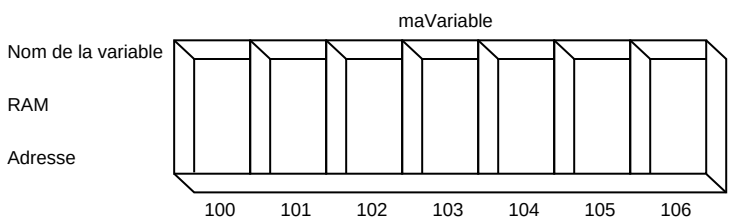


Figure 3.1
Représentation schématique de la mémoire.



RAM est l'acronyme de Random Access Memory (mémoire à accès direct). Lorsque vous lancez un programme, il est chargé du disque vers la RAM. La mémoire vive est également le siège de toutes les variables du programme. En programmation, le terme mémoire désigne le plus souvent la mémoire vive.

Généralités sur la mémoire

En C++, vous devez attribuer un type à chaque variable que vous définissez. Il peut s'agir d'une variable de type entier, d'un nombre à virgule flottante, d'un caractère, etc. Grâce à ces informations, le compilateur détermine l'espace à attribuer à la variable en mémoire, ainsi que le type des données qu'elle pourra contenir. Cela permet aussi au compilateur de vous avertir ou de produire un message d'erreur si vous tentez, par inadvertance, de stocker une valeur d'un type erroné dans votre variable (cette caractéristique de certains langages de programmation s'appelle *typage fort*).

Chaque emplacement de la RAM occupe un octet. Si le type de variable créé a une taille de 4 octets, celle-ci occupera donc quatre emplacements en mémoire. Le type de la variable (`int`, par exemple) indique au compilateur combien d'espace mémoire (ou d'emplacements) réserver dans la RAM.

À une époque, les programmeurs devaient impérativement comprendre la notion de bits et d'octets, qui sont en fait les unités de base de stockage. Bien que les langages actuels permettent de faire abstraction de ces détails, il demeure intéressant de comprendre comment sont stockées les données. Pour en savoir plus, reportez-vous à l'Annexe A.

Taille des entiers

D'un ordinateur à un autre, les tailles occupées par les différents types de variables peuvent être différentes, mais elles seront toujours les mêmes sur un ordinateur donné. Ainsi, un entier peut occuper deux octets sur une machine ou quatre sur une autre, mais cette propriété ne variera jamais sur l'une ou l'autre de ces machines.

Les caractères simples, comme les lettres, les chiffres ou les symboles, sont stockés dans des variables de type `char`, qui occupent généralement un octet.

Pour les nombres entiers les plus petits, on peut employer une variable de type `short` qui, sur la plupart des ordinateurs, occupera deux octets, alors qu'une variable de type `long` occupe généralement quatre octets. Si vous n'indiquez pas le mot-clé `short` ou `long`, la taille d'un entier varie de deux à quatre octets selon les ordinateurs.

En fait, contrairement à ce que l'on pourrait attendre, le langage ne le précise pas exactement. Tout ce que nous savons est que la taille d'un entier `short` doit être inférieure ou égale à celle d'un `int` qui, à son tour, doit être inférieure ou égale à celle d'un entier `long`.

Ceci dit, vous travaillez sûrement sur un ordinateur ayant des `short` sur deux octets et des `int` et des `long` de quatre octets.

La taille d'un entier est déterminée par le processeur (16 bits, 32 bits ou 64 bits) et le compilateur que vous utilisez. Sur une machine 32 bits (Pentium) utilisant des compilateurs modernes, les entiers occupent *quatre* octets.



Lorsque vous écrivez des programmes, ne faites jamais de supposition sur la taille mémoire utilisée pour représenter un type particulier.

Pour déterminer la taille exacte des types pris en charge par votre ordinateur, compilez et exécutez le programme du Listing 3.1.

Listing 3.1 : Déterminer la taille des types de variables sur votre ordinateur

```
 1:  #include <iostream>
 2:
 3:  int main()
 4:  {
 5:     using std::cout;
 6:
 7:     cout << "La taille d'un entier est :\t\t"
 8:        << sizeof(int)    << " octets.\n";
 9:     cout << "La taille d'un entier court est :\t"
10:        << sizeof(short)  << " octets.\n";
11:     cout << "La taille d'un entier long est :\t"
12:        << sizeof(long)   << " octets.\n";
13:     cout << "La taille d'un type char est :\t\t"
14:        << sizeof(char)   << " octets.\n";
15:     cout << "La taille d'un type float est :\t\t"
16:         << sizeof(float)  << " octets.\n";
17:     cout << "La taille d'un type double est :\t"
18:        << sizeof(double) << " octets.\n";
19:     cout << "La taille d'un booleen est :\t\t"
20:        << sizeof(bool)   << " octets.\n";
21:
22:     return 0;
23:  }
```

Ce programme produit le résultat suivant :

```
La taille d'un entier est :        4 octets.
La taille d'un entier court est :  2 octets.
La taille d'un entier long est :   4 octets.
La taille d'un type char est :     1 octets.
La taille d'un type float est :    4 octets.
La taille d'un type double est :   8 octets.
La taille d'un booleen est :       1 octets.
```



Sur votre ordinateur, le nombre d'octets annoncé peut être différent !

Le Listing 3.1 ne présente pas de difficultés majeures. Certaines lignes ont été scindées en deux pour des raisons d'édition. Par exemple, les lignes 7 et 8 pourraient normalement tenir sur une seule ligne. Le compilateur ignore les blancs (espace, tabulation, saut de ligne) et considère ces instructions comme une seule ligne. C'est pourquoi vous devez terminer la plupart des lignes par un signe ";".

Les lignes 7 à 20 font appel à l'opérateur `sizeof`, qui est utilisé comme une fonction. Il renvoie la taille de l'objet qui lui a été passé en paramètre. Par exemple, à la ligne 8, le mot clé `int` est passé à `sizeof`. Vous verrez plus loin dans ce chapitre que `int` permet de décrire une variable entière standard. Si l'on utilise `sizeof` sur un Pentium 4 sous Windows XP, la taille d'une valeur `int` équivaut à celle d'un entier `long`, soit quatre octets.

Les autres lignes du Listing 3.1 affichent les tailles des autres types de données. Vous en saurez plus sur les valeurs que ces types de données peuvent stocker et ce qui les différencie un peu plus loin.

signed et *unsigned*

Toutes les variables entières ont deux variantes : signée (`signed`) ou non signée (`unsigned`). En effet, on a parfois besoin de nombre négatifs, parfois non. Les entiers déclarés sans le mot-clé `unsigned` sont considérés comme signés. Les entiers signés sont soit négatifs, soit positifs, alors que les entiers non signés sont toujours positifs.

Les entiers, qu'ils soient signés ou non, occupent le même espace mémoire. De ce fait, une partie de la place de stockage d'un entier signé sera donc utilisée pour indiquer s'il est positif ou négatif. La valeur la plus élevée que vous pouvez stocker dans un entier non signé correspond donc au double du nombre positif le plus élevé contenu dans une variable entière signée.

Par exemple, si un entier `short` est stocké sur deux octets, un entier `short` non signé peut contenir une valeur comprise entre 0 et 65 535. Pour un entier `short` signé, la moitié des valeurs admises correspond à des nombres négatifs. L'intervalle autorisé se situe donc entre –32 768 et +32 767.

Types fondamentaux

C++ dispose de plusieurs types prédéfinis, qui peuvent être divisés en types entiers (voir plus haut), en types flottants et en type caractère.

Les variables à virgule flottante peuvent être exprimées sous la forme de fractions, ce sont des nombres réels. Les variables de type caractère n'occupent généralement qu'un seul octet et servent souvent à stocker les 256 caractères et symboles du jeu ASCII standard et des jeux ASCII étendus (en vigueur dans certaines langues).



Le jeu de caractères ASCII est un ensemble normalisé de lettres, de nombres et de symboles reconnu par tous les ordinateurs. ASCII est l'acronyme de American Standard Code for Information Interchange. Cette norme de codage est prise en charge par la plupart des systèmes d'exploitation, bien que nombre d'entre eux prennent également en charge d'autres jeux de caractères internationaux.

Les types de variables utilisées en C++ sont décrits dans le Tableau 3.1. En face du libellé du type C++ figurent la traduction française ainsi que sa taille et les valeurs qui peuvent être stockées. Ces valeurs sont déterminées par la taille des types de variables, conformément à la sortie produite par le programme du Listing 3.1 ; consultez son résultat pour voir si vos types de variables sont de la même taille. Il est très probable que ce sera le cas, à moins que vous n'utilisiez un ordinateur ayant un processeur à 64 bits.

Tableau 3.1 : Types de variables

Type C++	*Traduction française*	*Taille*	*Valeurs*
`bool`	booléen	1 octet	`true` (vrai) ou `false` (faux)
`unsigned short int`	entier court non signé	2 octets	de 0 à 65 535
`short int`	entier court	2 octets	de –32 768 à 32 767
`unsigned long int`	entier long non signé	4 octets	de 0 à 4 294 967 295
`long int`	entier long	4 octets	de –2 147 483 648 à 2 147 483 647
`int`	entier (16 bits)	2 octets	de –32 768 à 32 767
`int`	entier (32 bits)	4 octets	de –2 147 483 648 à 2 147 483 647
`unsigned int`	entier non signé (16 bits)	2 octets	de 0 à 65 535
`unsigned int`	entier non signé (32 bits)	2 octets	de 0 à 4 294 967 295
`char`	texte	1 octet	256 caractères
`float`	nombre décimal	4 octets	de 1,2e–38 à 3,4e38
`double`	nombre double	8 octets	de 2,2e–308 à 1,8e308



La taille des variables peut varier en fonction du compilateur et du processeur de votre ordinateur. En fait, le Tableau 3.1 et le listing qui précède se complètent, car les valeurs ont été extraites de la même configuration. Si le programme affiche un résultat différent, reportez-vous au manuel de votre compilateur.

Définition d'une variable

Pour l'instant, vous avez vu la création et l'utilisation de plusieurs variables. Il est temps maintenant d'apprendre à créer les vôtres.

Pour créer ou *définir* une variable, vous devez indiquer son type suivi de un ou plusieurs espaces, puis le nom de la variable suivi d'un point-virgule. Un nom de variable peut être composé de n'importe quelle combinaison de lettres, de chiffres et de symboles, mais ne peut pas contenir d'espace. Exemple : `x`, `J23qrsnf`, `monAge`. Le nom de la variable doit refléter son rôle, ce qui permet de suivre plus aisément le déroulement du programme. L'instruction suivante définit la variable entière `monAge` :

```
int monAge;
```



Lorsque vous déclarez une variable, la mémoire correspondante est allouée (réservée) à cette variable. La valeur de cette variable sera alors toute valeur se trouvant à cet emplacement mémoire à ce moment. Vous verrez bientôt comment affecter une nouvelle valeur à cet emplacement mémoire.

Évitez d'utiliser des noms barbares qui n'évoquent rien comme `J23qrsnf` et réservez les noms d'une seule lettre comme `x` ou `i`, à des variables qui ne seront utilisées qu'un bref instant. Les noms expressifs comme `Total` ou `monAge` sont bien plus faciles à comprendre et à mettre à jour.

Nous allons illustrer ces propos par deux exemples. Pour cela, tapez les deux blocs d'instructions suivants :

```
Exemple 1
int main()
{
     unsigned short x;
     unsigned short y;
     unsigned short z;
     z = x * y;
     return 0;
}
Exemple 2
int main()
{
     unsigned short Largeur;
     unsigned short Longueur;
     unsigned short Surface;
     Surface = Largeur * Longueur;
     return 0;
}
```



Si vous compilez ces programmes, votre compilateur vous signalera que ces valeurs ne sont pas initialisées. Nous allons bientôt voir comment résoudre ce problème.

Il est évident que le second programme est plus facile à comprendre. En outre, les noms longs donnent un sens au code et facilitent d'éventuelles modifications.

Respect de la casse des caractères

Le langage C++ fait la différence entre les majuscules et les minuscules, ce qui signifie qu'une variable appelée `age` est différente d'une variable appelée `Age`, qui est elle-même différente de la variable `AGE`.



Certains compilateurs permettent de désactiver cette fonctionnalité. Cette pratique est déconseillée car vous risquez de ne pas pouvoir compiler vos programmes avec d'autres compilateurs et d'en rendre la lecture plus difficile pour les autres programmeurs.

Conventions de nommage

Il existe plusieurs conventions pour les noms de variables. Peu importe celle que vous choisirez, mais il convient de rester homogène dans l'attribution des noms tout au long du programme de manière à faciliter la lecture du code par d'autres programmeurs.

Certains programmeurs détestent les majuscules et tapent tout le code en minuscules. Si le nom de la variable requiert deux éléments (par exemple, `sous total`), il est possible de les séparer à l'aide d'un caractère de soulignement, ou de les accoler. Exemple : `sous_total` ou `sousTotal`. La présence d'une lettre majuscule au milieu du nom de la variable facilite la lecture ; elle fait penser à une bosse et c'est la raison pour laquelle on l'appelle *camel case*.

L'utilisation des caractères de soulignement ne fait pas l'unanimité. Certains estiment que le code source est plus facile à lire, alors que d'autres pensent que la touche n'est pas bien située sur le clavier. Dans cet ouvrage, nous avons adopté la notation "dromadaire". Pour une meilleure lisibilité du texte, tous les éléments du nom (sauf le premier) commencent par une majuscule. Exemple : `monAge`, `totGeneral`, etc. Notez que nous n'utilisons pas les caractères accentués, la plupart des versions ne les prenant pas en charge.

En programmation avancée, on rencontre souvent la notation dite hongroise. Cette pratique consiste à utiliser un préfixe correspondant au type de la variable. Par exemple, les variables entières peuvent commencer par un *i* minuscule. Les variables de type `long` peuvent commencer par un *l*. Bien entendu, il existe d'autres préfixes signalant différentes constructions en C++ pour les pointeurs, les constantes, etc., que nous verrons par la suite.



L'origine du nom de la notation hongroise vient de son inventeur, un hongrois du nom de Charles Simonyi, de Microsoft. Vous pouvez trouver sa monographie originale à l'adresse : http://www.strangecreations.com//library/c/naming.txt.

Microsoft a abandonné récemment cette notation, et recommande de ne *pas* l'utiliser en C#. Cette recommandation est également valable pour C++.

Mots-clés

Certains mots sont réservés par C++ et ne peuvent donc pas être utilisés comme noms de variables. Il s'agit de mots-clés ayant une signification particulière pour le compilateur C++, par exemple `if`, `while`, `for` et `main`. Vous en trouverez une liste complète au Tableau 3.2 ainsi qu'à l'Annexe B. Il est possible que votre compilateur dispose d'autres mots réservés : vous devrez consulter son manuel pour en avoir la liste complète.

Tableau 3.2 : Les mots-clés de C++

`asm`	`else`	`new`	`this`
`auto`	`enum`	`operator`	`throw`
`bool`	`explicit`	`private`	`true`
`break`	`export`	`protected`	`try`
`case`	`extern`	`public`	`typedef`
`catch`	`false`	`register`	`typeid`
`char`	`float`	`reinterpret_cast`	`typename`
`class`	`for`	`return`	`union`
`const`	`friend`	`short`	`unsigned`
`const_cast`	`goto`	`signed`	`using`
`continue`	`if`	`sizeof`	`virtual`
`default`	`inline`	`static`	`void`
`delete`	`int`	`static_cast`	`volatile`
`do long`	`struct`	`wchar_t`	
`double`	`mutable`	`switch`	`while`
`dynamic_cast`	`namespace`	`template`	

Les mots suivants sont réservés :

`And`	`bitor`	`not_eq`	`xor`
`and_eq`	`compl`	`or`	`xor_eq`
`bitand`	`not`	`or_eq`	

Faire	*Ne pas faire*
● Définir une variable en indiquant son type et son nom. ● Utiliser des noms de variables significatifs. ● Ne pas oublier que C++ respecte la casse des caractères. ● Tenir compte de l'espace occupé par la variable en mémoire en fonction de son type, pour affecter des valeurs adéquates.	● Utiliser des mots-clés de C++ comme noms de variables. ● Présumer du nombre d'octets utilisés pour le stockage d'une variable. ● Utiliser des variables non signées pour stocker des nombres négatifs.

Création de plusieurs variables à la fois

Vous pouvez créer plusieurs variables de type identique dans la même instruction en séparant les noms par des virgules :

```
unsigned int monAge, monPoids;       // 2 vars entières non signées
long int surface, largeur, longueur; // 3 variables entières longues
```

Comme vous pouvez le constater, `monAge` et `monPoids` sont des variables entières non signées car il est inconcevable qu'un âge ou un poids soient négatifs. Sur la deuxième ligne, les variables `surface`, `largeur` et `longueur` sont de type `long`. Une même instruction de définition ne peut comprendre des types de variables différents.

Affectation de valeurs aux variables

Pour affecter une valeur à une variable, on utilise l'*opérateur d'affectation* (=). Dans l'expression suivante, on affecte la valeur 5 à la variable `largeur` :

```
unsigned short largeur;
largeur = 5;
```



`long` est un raccourci pour `long int`, comme `short` pour `short int`.

Il est possible de combiner les instructions de création d'une variable à celles d'affectation d'une valeur. Vous pouvez par exemple combiner ces deux étapes pour la variable `largeur` en écrivant :

```
unsigned short largeur = 5;
```

L'initialisation ressemble beaucoup à l'affectation précédente. Les différences entre les deux opérations sont mineures en ce qui concerne les entiers. Dans ce chapitre, vous allez apprendre que certaines variables doivent être initialisées, car il n'est plus possible de leur attribuer de valeur par la suite.

On peut également initialiser plusieurs variables simultanément. L'instruction suivante, par exemple, crée deux variables du type `long` et les initialise :

```
long largeur = 5, hauteur = 7;
```

Vous pouvez mélanger des définitions et des initialisations :

```
int monAge = 39, tonAge, sonAge = 40;
```

Dans cet exemple, trois variables entières sont créées, mais seules la première et la dernière sont initialisées.

Vous pouvez compiler le fichier source du Listing 3.2. Ce programme calcule la surface d'un rectangle, puis affiche le résultat à l'écran.

Listing 3.2 : Utilisation de variables

```
 1:  // Démonstration de variables
 2:  #include <iostream>
 3:
 4:  int main()
 5:  {
 6:     using std::cout;
 7:     using std::endl;
 8:
 9:     unsigned short int Largeur = 5, Longueur;
10:     Longueur = 10;
11:
12:     // Crée un entier court non signé
13:     // et l'initialise avec le résultat de la
```

```
13a:     // multiplication de Largeur par Longueur
 14:     unsigned short int Surface  = (Largeur * Longueur);
 15:
 16:     cout << "Largeur :" << Largeur << endl;
 17:     cout << "Longueur : "  << Longueur << endl;
 18:     cout << "Surface : " << Surface << endl;
 19:     return 0;
 20:  }
```

Ce programme produit le résultat suivant :

```
Largeur : 5
Longueur : 10
Surface : 50
```

Comme vous l'aurez constaté dans le précédent listing, à la ligne 2, l'instruction `include` insère la bibliothèque `iostream`, ce qui permet à `cout` de fonctionner. Le programme commence à la ligne 4 par la fonction `main()`. Les lignes 6 et 7 définissent `cout` et `endl` comme faisant partie de l'espace de nom standard (`std`).

La ligne 9 définit les premières variables. `Largeur` est un entier court non signé, initialisé à 5. En revanche, la variable `Longueur` est définie et non initialisée, elle reçoit la valeur 10 à la ligne 10.

La ligne 14 définit l'entier court non signé `Surface`, qui reçoit le produit de `Largeur` par `Longueur`. Les différentes valeurs sont ensuite affichées à l'écran (lignes 16 à 18). Le mot-clé `endl` permet de passer à la ligne.

Création d'alias avec *typedef*

À la longue, l'instruction `unsigned short int` peut devenir contraignante. Pourtant, elle présente l'avantage d'éviter les erreurs de type lors de la compilation. Le C++ permet de créer un alias en utilisant le mot-clé `typedef` (qui signifie *définition de type*).

En réalité, ce mot-clé permet de créer un synonyme, ce qui est différent de la création d'un nouveau type (voir Chapitre 6). `typedef` est suivi du nom du type existant puis de l'alias et d'un point-virgule. Exemple :

```
typedef unsigned short int USHORT
```

Le nouveau nom `USHORT` fait référence au type entier court non signé et peut être utilisé chaque fois que vous auriez écrit `unsigned short int`. Le Listing 3.3 ressemble au Listing 3.2, mais nous avons utilisé le nouveau type `USHORT`.

Listing 3.3 : Utilisation d'un alias de type

```
 1:  // Démonstration de l'utilisation du mot clé typedef
 2:  #include <iostream>
 3:
 4:  typedef unsigned short int USHORT;   //définition d'alias
 5:
 6:  int main()
 7:  {
 8:
 9:     using std::cout;
10:     using std::endl;
11:
12:     USHORT  Largeur = 5;
13:     USHORT Longueur;
14:     Longueur = 10;
15:     USHORT Surface  = Largeur * Longueur;
16:     cout << "Largeur :" << Largeur << endl;
17:     cout << "Longueur : "  << Longueur << endl;
18:     cout << "Surface : " << Surface <<endl;
19:     return 0;
20:  }
```

Ce programme produit le résultat suivant :

```
Largeur : 5
Longueur : 10
Surface : 50
```



L'astérisque (*) représente une multiplication.

À la ligne 4, `USHORT` est défini comme synonyme de `unsigned short int`. Par ailleurs, le programme est pratiquement identique au Listing 3.2 et la sortie est la même.

short ou *long* : que choisir ?

Les programmeurs qui débutent en C++ hésitent souvent entre ces deux types. La solution est simple : il est préférable de choisir le type `long` lorsque le contenu risque d'excéder la taille maximale autorisée pour une variable de type `short`.

Comme le montre le Tableau 3.1, les entiers courts non signés, supposés être d'une taille de deux octets, acceptent une valeur maximale de 65 535. Pour les entiers signés courts,

cette valeur est répartie entre les valeurs positives et négatives, et leur valeur maximale est donc la moitié de celle d'un entier court non signé.

En revanche, les entiers longs non signés prennent en charge une valeur comprise entre 0 et 4 294 967 295. Si vous envisagez de gérer une valeur supérieure, vous avez le choix entre le type `float` et le type `double`, au détriment de la précision numérique. Sur la plupart des ordinateurs, seules les 7 ou 9 premières positions sont prises en compte, ce qui signifie que le résultat sera arrondi après ces chiffres.

Les variables courtes occupent moins de mémoire mais, de nos jours, la mémoire n'est pas chère et la vie est courte, n'hésitez donc pas à utiliser des variables de type `int` qui occuperont probablement quatre octets sur votre machine.

Rebouclage des entiers non signés

Les entiers non signés acceptent une valeur limite. Qu'en est-il si vous la dépassez ?

Quand la valeur maximale d'une variable entière non signée est atteinte, le contenu de cette dernière est remis à zéro, un peu comme le compteur d'une voiture. Examinez le Listing 3.4.

Listing 3.4 : Incrémentation de un de la valeur maximale autorisée pour un entier *short* non signé

```
 1:  #include <iostream>
 2:  int main()
 3:  {
 4:     using std::cout;
 5:     using std::endl;
 6:
 7:     unsigned short int Nombre;
 8:     Nombre = 65535;
 9:     cout << "Nombre :" << Nombre << endl;
10:     Nombre++;
11:     cout << "Nombre :" << Nombre << endl;
12:     Nombre++;
13:     cout << "Nombre :" << Nombre << endl;
14:     return 0;
15:  }
```

Ce programme produit le résultat suivant :

```
Nombre :65535
Nombre :0
Nombre :1
```

À la ligne 7, `Nombre` est une variable de type entier court non signé qui, sur un Pentium 4 sous Windows XP, est d'une longueur de deux octets et peut donc contenir une valeur comprise entre 0 et 65 535. À la ligne suivante, cette variable reçoit la valeur maximale et le résultat s'affiche correctement (ligne 9).

À la ligne 10, la variable est incrémentée de un, comme le montre le symbole ++. Au passage, vous pouvez constater que le nom de C++ semble indiquer qu'il s'agit d'un incrément du langage C. La valeur de `Nombre` est donc égale à 65 536, valeur refusée pour un entier court non signé. La variable est remise à zéro et son contenu apparaît à l'écran.

À la ligne 12, `Nombre` est incrémenté de nouveau. Sa nouvelle valeur (1) s'affiche.

Rebouclage des entiers signés

Un entier signé se distingue d'un entier non signé puisque la moitié des valeurs admises sont négatives. Au lieu de vous représenter un compteur de voiture, vous pourriez imaginer une horloge comme celle de la Figure 3.2, où les chiffres augmentent dans les sens des aiguilles, et diminuent dans le sens inverse. Ils se croisent en bas du cadran (à 6 heures).

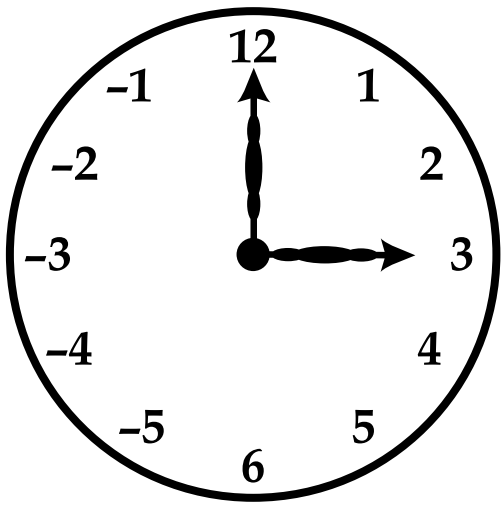


Figure 3.2
Si les pendules utilisaient des nombres signés.

Lorsque vous ne disposez plus de nombres positifs, vous atteignez le plus grand nombre négatif puis décomptez jusqu'à la valeur 0.

Dans le programme du Listing 3.5, le nombre positif maximal est incrémenté de un.

Listing 3.5 : Incrémentation de la valeur maximale admise pour un entier signé

```
 1:  #include <iostream>
 2:  int main()
 3:  {
 4:     short int Nombre;
 5:     Nombre = 32767;
 6:     std::cout << "Nombre :" << Nombre << std::endl;
 7:     Nombre++;
 8:     std::cout << "Nombre :" << Nombre << std::endl;
 9:     Nombre++;
10:     std::cout << "Nombre :" << Nombre << std::endl;
11:     return 0;
12:  }
```

Ce programme produit le résultat suivant :

```
Nombre : 32767
Nombre : -32768
Nombre : -32767
```

À la ligne 3, `Nombre` est déclaré comme entier court signé (par défaut, tout entier court est signé, pour définir un entier non signé, vous devez écrire explicitement le mot-clé `unsigned`). Le programme fonctionne sensiblement comme le précédent. Bien entendu, les résultats obtenus sont différents. Pour bien assimiler cet exemple, vous devez comprendre comment les nombres signés sont représentés sous forme binaire dans un entier d'une longueur de deux octets.

Comme pour un entier non signé, l'entier signé reboucle de la plus haute valeur positive à la plus haute valeur négative.

Variables caractère

Les variables caractère (type `char`) occupent généralement 1 octet et acceptent jusqu'à 256 valeurs (voir Annexe C). Ce type `char` peut être interprété comme un nombre court (0 à 255) ou comme un membre du jeu de caractères ASCII. Le jeu de caractères ASCII et son équivalent ISO permettent de coder toutes les lettres, les chiffres et les signes de ponctuation.



Les ordinateurs sont incapables de traiter des lettres, des signes de ponctuation ou des phrases. Ils ne comprennent que les nombres. En fait, leur rôle se limite à examiner une jonction de câbles particulière afin de déterminer si une charge

électrique suffisante peut y être détectée. Cette charge est alors représentée en interne comme un `1`, ou comme un `0` si elle est absente. En regroupant l'ensemble de ces 0 et 1, l'ordinateur est capable de générer des structures interprétées comme des nombres, qui à leur tour correspondent à des lettres ou à des signes de ponctuation.

En code ASCII, la valeur `97` correspond à la lettre "a" minuscule. Les majuscules, les minuscules, les chiffres et les signes de ponctuation sont associés aux valeurs ASCII comprises entre 1 et 128. Les 128 valeurs restantes sont réservées au constructeur informatique, bien que le jeu IBM étendu soit devenu une norme utilisée par de nombreux programmeurs.



ASCII se prononce "A-ski".

Caractères et nombres

Tout caractère affecté à une variable `char` (comme `'a'`) correspond en fait à un nombre entre 0 et 255. Le compilateur est capable d'identifier, de coder et de décoder tout caractère, nombre ou signe de ponctuation figurant entre des guillemets simples.

La relation valeur/lettre est arbitraire. Il s'agit d'une convention prise en charge par le clavier, le compilateur et l'écran. Attention : la valeur `5` et le caractère `'5'` ne sont pas égaux, ce dernier ayant une valeur ASCII de `53`. Cette notion est illustrée dans le Listing 3.6.

Listing 3.6 : Impression de caractères à partir de leurs cSCII

```
1:  #include <iostream>
2:  int main()
3:  {
4:     for (int i = 32; i<128; i++)
5:        std::cout << (char) i;
6:     return 0;
7:  }
```

Ce programme produit le résultat suivant :

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Ce programme permet d'afficher les valeurs caractères des codes ASCII compris entre 32 et 127. Pour accomplir cette tâche, ce listing utilise une variable entière, `i`, à la ligne 4. En ligne 5, le nombre de la variable `i` est obligé de s'afficher sous forme de caractère.

On aurait aussi pu utiliser une variable caractère, comme dans le Listing 3.7, pour arriver au même résultat.

Listing 3.7 : Impression de caractères à partir de leurs codes ASCII – version 2

```
1:  #include <iostream>
2:  int main()
3:  {
4:     for (unsigned char i = 32; i<128; i++)
5:        std::cout <<  i;
6:     return 0;
7:  }
```

Comme vous le pouvez le constater à la ligne 4, on utilise ici un caractère non signé. Une variable de caractère étant utilisée à la place d'une variable numérique, le `cout` de la ligne 5 sait afficher la valeur du caractère.

Caractères spéciaux pour l'impression

Le compilateur C++ reconnaît certains caractères spéciaux de formatage. Le Tableau 3.3 regroupe les codes de mise en forme les plus courants. Pour les insérer dans un code source, il suffit de les faire précéder d'une barre oblique inverse (appelée caractère d'échappement). Pour insérer une tabulation, tapez la ligne suivante :

```
char carTab = '\t';
```

Cet exemple déclare une variable de type `char` (`carTab`) qui est initialisée avec la valeur `\t` reconnue comme une tabulation. Les caractères de formatage sont utilisés sur n'importe quel périphérique de sortie : écran, imprimante et fichier disque.

Le caractère d'échappement (\) modifie la signification du caractère qu'il précède. Par exemple, si vous tapez `n`, vous affichez la 14e lettre de l'alphabet, alors que précédé du caractère d'échappement, il correspond à une nouvelle ligne.

Tableau 3.3 : Caractères d'échappement

Caractère	*Signification*
`\a`	alerte sonore
`\b`	retour arrière

Tableau 3.3 : Caractères d'échappement *(suite)*

Caractère	*Signification*
`\f`	saut de page
`\n`	nouvelle ligne
`\r`	retour chariot
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\'`	guillemet simple
`\"`	guillemet double
`\?`	point d'interrogation
`\\`	barre oblique inverse
`\000`	notation octale
`\xhhh`	notation hexadécimale

Constantes

Comme les variables, les *constantes* sont des emplacements de stockage d'information. Mais, à la différence des variables, elles ne sont pas modifiables : vous devez donc initialiser une constante lorsque vous la déclarez et vous ne pourrez plus ensuite lui affecter de nouvelle valeur.

C++ distingue deux types de constantes : les constantes littérales et les constantes symboliques.

Constantes littérales

Une *constante littérale* est une valeur apparaissant directement dans le code du programme au moment où l'on en a besoin. Exemple :

```
int monAge = 32;
```

`monAge` est une variable de type `int`, alors que `32` est une constante littérale. Vous ne pouvez pas affecter de valeur à `32` et sa valeur ne peut pas être modifiée.

Constantes symboliques

Comme une variable, une *constante symbolique* est représentée par un nom. Toutefois, son contenu ne peut être modifié après son initialisation.

Supposons que vous ayez déclaré deux variables de type entier `Eleves` et `Classes` et que souhaitiez connaître le nombre d'élèves d'une école. Si vous savez que chaque classe est formée de 15 élèves, le calcul se présentera ainsi :

```
Eleves = Classes * 15;
```

Dans cet exemple, `15` est une constante littérale. Votre programme serait plus facile à lire et à mettre à jour si cette valeur était représentée par une constante symbolique :

```
Eleves = Classes * elevesParClasse
```

Si vous décidez plus tard de modifier le nombre d'élèves par classe, il vous suffira de changer la valeur d'initialisation de la constante `elevesParClasse` : vous n'aurez pas besoin de modifier toutes les instructions qui utilisent cette valeur.

Il existe deux façons de déclarer une constante symbolique en C++. La plus ancienne, désormais obsolète, consiste à utiliser la directive `#define` du préprocesseur. La seconde, plus appropriée, consiste à utiliser le mot-clé `const`.

Définition de constantes avec *#define*

De nombreux programmes faisant appel à la directive `#define`, vous devez bien en comprendre le fonctionnement. Pour définir une constante de cette manière (obsolète), vous pouvez taper :

```
#define elevesParClasse 15
```

La constante `elevesParClasse` n'a pas de type particulier (`int`, `char`, etc.). En fait, le préprocesseur effectuera une simple substitution de texte : à chaque fois qu'il rencontrera le mot `elevesParClasse` dans le texte du programme, il le remplacera `15`.

Le préprocesseur s'exécutant avant le compilateur, ce dernier ne verra donc jamais votre constante, mais uniquement le nombre 15.



Même si `#define` semble très simple à utiliser, il est préférable de l'éviter car la norme C++ l'a déclaré obsolète.

Définition de constantes à l'aide de *const*

Bien que la directive `#define` fonctionne, le mot-clé `const` permet de créer des constantes C++ plus proprement :

```
const unsigned short int elevesParClasse = 15;
```

Cet exemple crée également la constante symbolique `elevesParClasse` mais, cette fois-ci, elle est de type entier court non signé. Grâce à cette méthode, les erreurs sont moins fréquentes et le code peut être mis à jour facilement car le mot-clé `const` permet au compilateur de connaître le type de la constante et de vérifier qu'elle est correctement utilisée.



Il est impossible de modifier le contenu d'une constante pendant l'exécution d'un programme. Pour affecter une autre valeur à une constante, vous devez intervenir au niveau du fichier source, puis le recompiler.

Faire	*Ne pas faire*
● Vérifier que la valeur affectée à d'un entier de dépasse pas ses limites afin qu'il n'y ait pas de rebouclage. ● Attribuer des noms significatifs aux constantes.	● Utiliser des mots-clés comme noms de constantes. ● Utiliser la directive `#define` du préprocesseur pour déclarer des constantes. Préférer `const`.

Constantes énumérées

Ces constantes permettent de créer de nouveaux types, puis de définir des variables dont les valeurs seront limitées à un ensemble précis. Vous pourriez par exemple créer une énumération pour y stocker des couleurs. En ce cas, il faudrait déclarer l'énumération `COULEURS`, qui serait associée aux cinq valeurs `ROUGE`, `BLEU`, `VERT`, `BLANC` et `NOIR`.

La syntaxe consiste à écrire le mot-clé `enum`, suivi du nom du type, d'une accolade ouvrante, des valeurs admises séparées par des virgules, d'une accolade fermante et d'un point-virgule. Exemple :

```
enum COULEURS { ROUGE, BLEU, VERT, BLANC, NOIR };
```

Cette instruction effectue deux actions :

1. Elle crée le nom d'énumération `COULEURS`, c'est-à-dire un nouveau type.

2. Elle crée des constantes symbolique comme ROUGE, BLEU, VERT, etc. en les initialisant respectivement avec les valeurs 0, 1, 2, etc.

À chaque constante correspond une valeur entière. En l'absence d'indication contraire, la première constante sera initialisée à 0, la seconde à 1, etc. Vous pouvez toutefois affecter des valeurs explicites ; celles qui n'en n'ont pas recevront la valeur de la constante précédente, plus un. Ici, par exemple :

```
enum Couleurs { ROUGE=100, BLEU, VERT=500, BLANC, NOIR=700 };
```

ROUGE ayant la valeur 100, BLEU sera associée à la valeur 101 ; selon le même principe, BLANC vaudra 501.

Vous pouvez définir des variables du type COULEURS, mais elles ne pourront recevoir que des valeurs de l'énumération (ici, ROUGE, BLEU, VERT, BLANC ou NOIR).

Il faut bien réaliser que les variables énumérées sont généralement de type unsigned int et que les constantes de l'énumération sont, en fait, des valeurs entières. Il est toutefois très pratique de pouvoir utiliser des noms évocateurs pour ces valeurs lorsque l'on travaille sur des couleurs, des jours de la semaine, ou tout autre ensemble fini de valeurs. Le Listing 3.8 utilise une énumération pour représenter des jours :

Listing 3.8 : Exemple d'énumération

```
 1:  #include <iostream>
 2:  int main()
 3:  {
 4:     enum Jours { Dimanche, Lundi, Mardi,
 5:                  Mercredi, Jeudi, Vendredi, Samedi };
 6:
 7:     Jours aujourdhui;
 8:     aujourdhui = Lundi;
 9:
10:     if (aujourdhui == Dimanche || aujourdhui == Samedi)
11:        std::cout << "\nJ'adore les week-ends !\n";
12:     else
13:        std::cout << "\nAu boulot !\n";
14:
15:     return 0;
16:  }
```

Ce programme produit le résultat suivant :

```
Au boulot !
```

Aux lignes 4 et 5, on définit la constante énumérée `Jours` avec sept valeurs équivalant chacune à un entier en partant de 0 ; la valeur de `Lundi` est donc 1 (`Dimanche` valait 0).

La ligne 7 crée une variable de type `Jours` qui contiendra une valeur valide à partir de la liste des constantes énumérées définies aux lignes 4 et 5. La valeur `Lundi` est affectée à cette variable en ligne 8 et le contenu de la variable est testée en ligne 10.

La constante énumérée de la ligne 8 pourrait être remplacée par une série de constantes entières, comme dans le Listing 3.9.

Listing 3.9 : Le même programme utilisant des constantes entières

```
 1:  #include <iostream>
 2:  int main()
 3:  {
 4:     const int Dimanche = 0;
 5:     const int Lundi = 1;
 6:     const int Mardi = 2;
 7:     const int Mercredi = 3;
 8:     const int Jeudi = 4;
 9:     const int Vendredi = 5;
10:     const int Samedi = 6;
11:
12:     int aujourdhui;
13:     aujourdhui = Lundi;
14:
15:     if (aujourdhui == Dimanche || aujourdhui == Samedi)
16:        std::cout << "\nJ'adore les week-ends !\n";
17:     else
18:        std::cout << "\nAu boulot !\n";
19:
20:     return 0;
21:  }
```

Ce programme produit le résultat suivant :

```
Au boulot !
```



Plusieurs variables déclarées dans ce programme n'étant pas utilisées, votre compilateur risque de vous en avertir au moment de la compilation.

La sortie produite est identique à celle du Listing 3.8. Chacune des constantes (`Dimanche`, `Lundi`, etc.) a été ici définie explicitement et il n'existe pas de type énuméré `Jours`.

Les constantes énumérées présentent l'avantage d'être parfaitement claires, l'utilité du type énuméré `Jours` est évidente.

Questions-réponses

Q Si un entier court peut être remis à zéro en cas de dépassement de capacité, pourquoi ne pas toujours utiliser des entiers longs ?

R Tous les types d'entiers pourront renvoyer des valeurs erronées en cas de dépassement de capacité. Toutefois, les valeurs maximales acceptées seront de 65 535 pour un `unsigned short int` de deux octets, et de 4 294 967 295 pour un `unsigned long int` de quatre octets. Cependant, sur la plupart des machines, un entier `long` occupera davantage de mémoire (4 octets contre 2) et, dans ce cas, un programme contenant 100 variables de ce type occupera donc 200 octets de mémoire vive supplémentaires. Désormais ce problème n'est plus crucial, car les PC disposent maintenant de quantités de mémoire considérables.

Utiliser des types supérieurs à vos besoins peut également demander plus de traitement de la part du processeur.

Q Que se passe-t-il si j'affecte un nombre à virgule à un entier alors que j'aurai dû utiliser un `float` ?

Par exemple :

```
int unNombre = 5.4;
```

R Dans ce cas, un compilateur efficace enverra un avertissement, mais cette affectation est autorisée. Le nombre affecté sera tronqué : la valeur 5,4 deviendra la valeur entière 5, ce qui entraînera une perte d'information.

Q Pourquoi doit-on utiliser des constantes symboliques et ne pas se contenter des constantes littérale ?

R Si vous utilisez la même valeur en de nombreux endroits de votre programme, vous pourrez la modifier en une seule fois si elle est désignée par une constante symbolique. Par ailleurs une constante symbolique, de par son nom, est plus explicite. Il peut être difficile de comprendre pourquoi un nombre est multiplié par la constante 60, alors que cela deviendra évident s'il s'agit de la constante `secParMinute`.

Q Que se passe-t-il si j'affecte un nombre négatif à une variable non signée ?

Par exemple :

```
unsigned int unNombrePositif  = -1;
```

R Un bon compilateur renverra un avertissement, mais cette affectation est autorisée. La valeur sera alors considérée comme une suite de bit représentant un nombre non signé et sera affectée à la variable. Par exemple `-1` dont la représentation binaire est `11111111 11111111` (`0xFF` en code hexadécimal) sera interprété comme la valeur 65 535 non signée.

Q Est-il possible de travailler en C++ sans comprendre les structures binaires, l'arithmétique binaire et le système hexadécimal ?

R Oui, mais moins efficacement que si vous maîtrisez ces notions. Le langage C++ ne masque pas autant que d'autres langages les détails internes de l'ordinateur. Ceci peut être un avantage puisque vous disposez ainsi d'une bien plus grande marge de manœuvre, mais cela peut aussi entraîner des résultats non souhaités en cas d'erreur de programmation. Bien comprendre ces concepts permet d'en exploiter pleinement les possibilités. Les programmeurs C++ qui ne maîtrisent pas les bases du système binaire sont souvent déconcertés par certains résultats.

Testez vos connaissances

1. Quelle est la différence entre une variable entière et une variable à virgule flottante ?
2. Quelles sont les différences entre un `unsigned short int` et un `long int` ?
3. Pourquoi préférer les constantes symboliques aux constantes littérales ?
4. Pourquoi préférer le mot-clé `const` à la directive `#define` ?
5. Qu'est-ce qui caractérise un "bon" nom de variable ou un "mauvais" nom ?
6. Dans cette énumération, quelle est la valeur de `BLEU` ?

```
enum COLOR { BLANC, NOIR = 100, ROUGE, BLEU, VERT = 300 };
```

7. Parmi ces noms de variables, quels sont ceux qui sont bons et mauvais, quels sont ceux qui ne sont pas corrects ?
   a. `Age`
   b. `!ex`
   c. `R79J`
   d. `revenuTotal`
   e. `__Invalide`

Exercices

1. Quel doivent être les types des variables destinées à recevoir les informations suivantes ?
   a. Votre âge
   b. La surface de votre jardin
   c. Le nombre d'étoiles de la galaxie
   d. La pluviométrie moyenne de janvier
2. Proposez des noms de variables évocateurs pour ces informations.
3. Déclarez une constante pour `pi = 3.14159`.
4. Déclarez une variable de type `float`, puis initialisez-la avec la constante `pi`.



4

Expressions et instructions

Au sommaire de ce chapitre

- Les instructions
- Les blocs d'instructions
- Les expressions
- Les branchements conditionnels
- Les conditions logiques

Un programme est, à la base, un ensemble de commandes qui s'exécutent séquentiellement. Il est possible de modifier cette séquence et d'exécuter un jeu de commandes ou un autre selon qu'une condition particulière est ou n'est pas satisfaite.

Instructions

En C++, une instruction a pour rôle de contrôler la séquence d'exécution, d'évaluer une expression ou de ne rien faire (instruction nulle). Toutes les instructions C++, y compris l'instruction nulle, se terminent par un point-virgule.

L'une des instructions les plus courantes, l'affectation, ressemble à ceci :

```
x = a + b;
```

À la différence de l'algèbre, cette instruction ne signifie pas que `x` est égal à `a+b`, mais qu'il faut "affecter la valeur de la somme `a` plus `b` à `x`".

Cette instruction effectue deux opérations : elle ajoute `a` et `b`, puis affecte le résultat à `x` à l'aide de l'opérateur d'affectation (`=`). Bien qu'elle réalise deux opérations, il ne s'agit que d'une seule instruction, qui se termine donc par un point-virgule.



L'opérateur d'affectation attribue la valeur à droite du signe égal à la variable située à sa gauche.

Espace ou blanc

Un espace correspond à un caractère invisible (tabulation, espace, saut de ligne). On les appelle "caractères blancs", car ils sont invisibles à l'impression.

Généralement, les instructions ignorent les espaces. L'instruction d'affectation ci-dessus, par exemple, pourrait s'écrire ainsi :

```
x=a+b;
```

ou ainsi :

```
x                       =a
+           b           ;
```

Bien que correcte, cette dernière présentation n'est pas très convaincante. Les espaces sont destinés à rendre le programme plus lisible, non à l'éparpiller dans le listing. Dans tous les cas, le C++ propose, le programmeur dispose !

Blocs et instructions composées

Partout où vous pouvez placer une instruction simple, vous pouvez placer une instruction composée, appelée *bloc*. Un bloc commence par une accolade ouvrante et se termine par

une accolade fermante. Bien que chaque instruction du bloc doive se terminer par un point-virgule, le bloc lui-même se termine simplement par l'accolade fermante. Exemple :

```
{
temp = a;
a = b;
b = temp;
}
```

Ce bloc agit comme une seule instruction, en intervertissant les valeurs de `a` et de `b`.

Faire	*Ne pas faire*
● Terminer chaque instruction par un point-virgule. ● Utiliser judicieusement des espaces pour rendre le code plus lisible.	● Oublier d'insérer une accolade fermante correspondant à une accolade ouvrante.

Expressions

En C++, tout ce qui correspond à une valeur est une *expression*. On dit qu'une expression *renvoie* une valeur : `3 + 2`, par exemple, est une expression qui renvoie la valeur 5. Toutes les expressions sont des instructions.

Les expressions sont nombreuses et de formats différents. Voici quelques exemples :

```
3.2                 // renvoie la valeur 3,2
PI                  // constante "float" renvoyant la valeur 3,14
SecondesParMinute   // constante entière renvoyant 60
```

En admettant que `PI` et `SecondesParMinute` soient des constantes initialisées respectivement à 3,14 et à 60, ces trois instructions sont des expressions.

Voici une expression un peu plus complexe :

```
x = a + b;
```

Elle additionne `a` et `b`, affecte le résultat à `x`, mais renvoie aussi la valeur qui vient d'être affectée (celle de `x`) : cette instruction d'affectation est donc une expression.

N'importe quelle expression peut apparaître à droite d'un opérateur d'affectation, ce qui inclut donc l'instruction d'affectation que nous venons de voir. On peut donc également écrire :

```
y = x = a + b;
```

Cette ligne est évaluée dans l'ordre suivant :

1. Ajouter a à b.
2. Affecter le résultat de l'expression a + b à x.
3. Affecter le résultat de l'expression x = a + b à y.

Si a, b, x et y sont des entiers et que a et b valent respectivement 9 et 7, x et y recevront toutes les deux la valeur 16, comme le montre le Listing 4.1.

Listing 4.1 : Évaluation d'expressions

```
 1:  #include <iostream>
 2:  int main()
 3:  {
 4:     using std::cout;
 5:     using std::endl;
 6:
 7:     int a=0, b=0, x=0, y=35;
 8:     cout << "a: " << a << " b: " << b;
 9:     cout << " x: " << x << " y: " << y << endl;
10:     a = 9;
11:     b = 7;
12:     y = x = a+b;
13:     cout << "a: " << a << " b: " << b;
14:     cout << " x: " << x << " y: " << y << endl;
15:     return 0;
16:  }
```

Ce programme produit le résultat suivant : ,

```
a: 0 b: 0 x: 0 y: 35
a: 9 b: 7 x: 16 y: 16
```

La ligne 7 déclare et initialise quatre variables dont les valeurs sont affichées aux lignes 8 et 9. La ligne 10 affecte la valeur 9 à a, la ligne 11, la valeur 7 à b. La ligne 12 additionne a et b et affecte le résultat à x. En conséquence, l'expression (x = a+b) est évaluée pour produire une valeur (la somme de a et b), qui est à son tour affectée à y. Aux lignes 13 et 14, ces résultats sont confirmés par l'affichage des valeurs des quatre variables.

Opérateurs

Un opérateur est un symbole qui demande au compilateur d'effectuer une action. Les opérateurs agissent sur les opérandes qui sont des expressions en langage C++. Il existe plusieurs catégories d'opérateurs ; les deux premières que nous étudierons ici sont :

- les opérateurs d'affectation ;
- les opérateurs mathématiques.

Opérateurs d'affectation

Vous avez déjà vu l'opérateur d'affectation (=). Il transfère la valeur située à droite de l'opérateur dans l'opérande de gauche. L'expression suivante :

```
x = a + b;
```

affecte le résultat de l'addition à l'opérande `x`.

---

lvalues et rvalues

Tout opérande situé à gauche de l'opérateur d'affectation est appelé *lvalue* (*left value)*, alors que l'opérande situé à droite s'appelle *rvalue* (*right value*).

Toutes les *lvalues* sont des *rvalues*, mais l'inverse n'est pas toujours vrai. Une *rvalue* littérale, par exemple, n'est pas un *lvalue*. Vous pouvez donc écrire :

```
x = 5;
```

mais pas :

```
5 = x;
```

`x` peut être une *lvalue* ou une *rvalue*, alors que `5` ne peut être qu'une *rvalue*.



Les constantes sont des rvalues : leurs valeurs ne pouvant être modifiées, elles ne peuvent pas se trouver du côté gauche de l'opérateur d'affectation, ce qui signifie qu'elles ne peuvent pas être des lvalues.

Opérateurs mathématiques

L'autre catégorie d'opérateurs regroupe les opérateurs mathématiques. Il existe cinq opérateurs mathématiques : addition (`+`), soustraction (`-`), multiplication (`*`), division (`/`) et modulo (`%`).

Les opérateurs + et – fonctionnent comme l'on s'y attend. La multiplication utilise le signe astérisque (*) et la division la barre oblique. Les exemples suivants illustrent chacun de ces opérateurs. Dans chaque cas, le résultat est affecté à la variable `resultat`. Les commentaires à droite montrent la valeur attendue.

```
resultat = 56 + 32   // resultat = 88
resultat = 12 - 10   // resultat = 2
resultat = 21 / 7    // resultat = 3
resultat = 12 * 4    // resultat = 48
```

Problèmes de soustraction

La soustraction d'entiers non signés (`unsigned`) peut conduire à des résultats étonnants lorsque le résultat est un nombre négatif. Dans le chapitre précédent, nous avons abordé le dépassement de capacité d'une variable négative. Dans le Listing 4.2, nous allons soustraire d'un nombre non signé un autre nombre non signé de valeur supérieure.

Listing 4.2 : Soustraction d'entiers et dépassement de capacité

```
 1:  // Listing 4.2 - soustraction et
 2:  // dépassement de capacité
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     using std::cout;
 8:     using std::endl;
 9:
10:     unsigned int difference;
11:     unsigned int grandNombre = 100;
12:     unsigned int petitNombre = 50;
13:
14:     difference = grandNombre - petitNombre;
15:     cout << "La difference est : " << difference;
16:
17:     difference = petitNombre - grandNombre;
18:     cout << "\nMaintenant la difference est : " << difference <<endl;
19:     return 0;
20:  }
```

Ce programme produit le résultat suivant :

```
La difference est : 50
Maintenant la difference est : 4294967246
```

Le résultat de la soustraction effectuée à la ligne 14 s'affiche pour la première fois à la ligne 15. L'opérateur est appelé de nouveau à la ligne 17 mais, comme on a retranché un grand nombre d'un nombre plus petit, le résultat est négatif. Comme ce résultat est évalué (et affiché) comme un nombre non signé, il y a dépassement de capacité, comme on l'a expliqué au Chapitre 3. Pour plus de détails, consultez l'Annexe C.

Division entière et modulo

La division entière est celle que vous avez étudiée à l'école primaire. Si vous divisez 21 par 4 (21 / 4), la réponse est 5 (avec un reste).

Le cinquième opérateur mathématique vous est peut-être inconnu. Il s'agit de l'opérateur modulo (`%`) qui donne la valeur du reste de la division entière. Pour obtenir le reste de la division de 21 par 4, on calcule 21 modulo 4 (`21 % 4`), ce qui donne 1.

Extraire le reste peut être très utile pour, par exemple, afficher une instruction toutes les dix actions. En effet, si le reste de la division d'un nombre par 10 est égal à 0, ce nombre est un multiple de 10. Ainsi, `1 % 10` vaut 1, `2 % 10` vaut 2, etc., jusqu'à `10 % 10` qui vaut 0. `11 % 10` repasse à 1 et cette série se continue jusqu'au prochain multiple de 10, qui est 20 (`20 % 10` vaut à nouveau 0). Vous utiliserez cette technique avec les boucles au Chapitre 7.

Si je divise 5 par 3 j'obtiens 1. Qu'est-ce qui ne va pas ?

Si vous divisez un entier par un autre, vous obtenez un entier comme résultat.

Par conséquent `5 / 3` donne 1 (en réalité la réponse est 1 avec un reste de 2, pour obtenir le reste, faites `5 % 3` et vous obtiendrez 2).

Pour obtenir une valeur fractionnaire, utilisez des chiffres à virgule flottante (types `float`, `double` ou `long double`).

`5.0 / 3.0` vous donnera un résultat fractionnaire : 1,66667

Si, soit le diviseur, soit le dividende est une valeur à virgule flottante, le compilateur produit un quotient à virgule flottante. Toutefois, s'il est affecté à une *lvalue* entière, la valeur sera à nouveau tronquée.

Combinaison d'opérateurs d'affectation et d'opérateurs mathématiques

Il est courant d'ajouter une valeur à une variable, puis de renvoyer le résultat dans cette dernière. Par exemple, prenons la variable `age`, à laquelle nous allons ajouter 2 :

```
int age = 5;
int temp;
temp = age + 2;          // 5 + 2 dans temp
age = temp;              // affectation de temp à age
```

Les deux premières lignes créent la variable `age` ainsi qu'une variable temporaire. Comme on le voit à la troisième ligne, la valeur `age` se voit ajouter 2 et le résultat est affecté à `temp`. La ligne suivante, cette valeur est replacée dans `age`, afin de la mettre à jour.

Cette méthode est terriblement compliquée et un peu difficile à gérer. En C++, vous pouvez utiliser la même variable des deux côtés de l'opérateur d'affectation, ce qui permet d'écrire :

```
age = age + 2;
```

Cette expression est bien plus claire, même si elle n'est pas conforme à la notation algébrique. Le C++ l'interprète comme : "ajouter 2 à la variable `age`, puis affecter le résultat à cette même variable".

Il est également possible de faire plus court :

```
age += 2;
```

Le résultat est identique, mais peut-être un peu moins lisible.

Cette ligne utilise l'opérateur d'addition combiné à l'affectation (`+=`) qui ajoute la valeur droite à la valeur gauche puis affecte à nouveau le résultat à la valeur gauche. Si `age` valait 24 au départ, il vaudra donc 26 après cette instruction.

Il existe d'autres opérateurs combinés à l'affectation pour la soustraction (`-=`), la division (`/=`), la multiplication (`*=`) et le modulo (`%=`).

Incrémentation et décrémentation

La valeur que l'on ajoute (ou soustrait) le plus souvent est 1. En C++, augmenter une valeur de 1 est appelé *incrémentation*, diminuer une valeur de 1 est appelé *décrémentation*. Pour effectuer ces actions, le langage C++ propose deux opérateurs spéciaux.

L'opérateur d'incrémentation (++) augmente de 1 la valeur d'une variable, tandis que l'opérateur de décrémentation (--) la diminue de 1. Dans l'exemple ci-dessous, nous allons incrémenter la variable `compteur` :

```
compteur++;               // compteur est égal à compteur plus 1.
```

Ce qui revient à écrire :

```
compteur = compteur + 1;
```

ou encore :

```
compteur += 1;
```



Comme vous l'avez peut-être deviné, le nom C++ provient de l'application de l'opérateur d'incrémentation au nom de son prédécesseur, le langage C. L'idée est que le C++ est une version supérieure de C.

Préfixe et suffixe

Les opérateurs d'incrémentation et de décrémentation existent en deux versions : préfixe et suffixe. Un préfixe précède le nom de la variable (`++age`), alors que le suffixe le suit (`age++`).

Dans une instruction simple, vous pouvez utiliser l'un ou l'autre, mais dans une instruction complexe où vous incrémentez (ou décrémentez) une variable puis affectez le résultat à une autre variable, la nuance est importante.

Le principe du préfixe est que l'on effectue l'incrémentation *avant* de renvoyer la valeur alors que, dans le cas du suffixe, on incrémente *après* avoir renvoyé la valeur.

Un exemple permettra d'éclaircir ce concept. Supposons que `x` soit un entier dont la valeur est 5 et que vous utilisiez un opérateur d'incrémentation préfixe. Si vous écrivez :

```
int a = ++x;
```

le compilateur va incrémenter `x` (qui prendra la valeur 6) *avant* d'en affecter la valeur à `a`. Ces deux variables auront donc la même valeur 6.

Si vous utilisez ensuite l'opérateur suffixe pour écrire :

```
int b = x++;
```

le compilateur va attribuer la valeur de `x` (6) à `b`, puis incrémenter `x` qui prendra la valeur `7`. `b` vaut donc 6 et `x` vaut 7. Le Listing 4.3 est un exemple d'utilisation des préfixes et des suffixes.

Listing 4.3 : Opérateurs préfixe et suffixe

```
 1:  // Listing 4.3 - Utilisation des opérateurs
 2:  // d'incrémentation et de décrémentation
 3:  // en préfixe et suffixe
 4:  #include <iostream>
 5:  int main()
 6:  {
 7:     using std::cout;
 8:
 9:     int monAge = 39;      // initialise deux entiers
10:     int tonAge = 39;
11:     cout << "J'ai : " << monAge << " ans.\n";
12:     cout << "Tu as : " << tonAge << " ans\n";
13:     monAge++;            // suffixe
14:     ++tonAge;            // préfixe
15:     cout << "Un an plus tard...\n";
16:     cout << "J'ai : " << monAge << " ans.\n";
17:     cout << "Tu as : " << tonAge << " ans\n";
18:     cout << "Encore une annee de plus\n";
19:     cout << "J'ai : " << monAge++ << " ans.\n";
20:     cout << "Tu as : " << ++tonAge << " ans\n";
21:     cout << "Reaffichons le resultat\n";
22:     cout << "J'ai : " << monAge << " ans.\n";
23:     cout << "Tu as : " << tonAge << " ans\n";
24:     return 0;
25:  }
```

Ce programme produit le résultat suivant :

```
J'ai : 39 ans
Tu as : 39 ans
Un an plus tard...
J'ai : 40 ans
Tu as : 40 ans
Encore une annee de plus
J'ai : 40 ans
Tu as : 41 ans
Reaffichons le resultat
J'ai : 41 ans
Tu as : 41 ans
```

Les lignes 9 et 10 déclarent deux variables entières initialisées à 39. Leurs valeurs s'affichent aux lignes suivantes.

Aux lignes 13 et 14, `monAge` et `tonAge` sont incrémentées respectivement à l'aide d'un opérateur suffixe et d'un préfixe. Le résultat est égal à `40` (lignes 16 et 17).

À la ligne 19, `monAge` est incrémenté dans l'instruction d'affichage, à l'aide d'un opérateur suffixe ; l'incrémentation s'effectue donc *après* l'affichage qui produit à nouveau `40`, puis

la variable `monAge` est incrémentée. À la ligne 20, `tonAge` est incrémenté par préfixe, donc *avant* l'affichage qui produit 41.

Aux lignes 22 et 23, le résultat est réaffiché. Les deux valeurs sont égales puisque l'instruction d'incrémentation a été exécutée.

Priorité des opérateurs

Dans l'instruction composée suivante :

```
x = 5 + 3 * 8;
```

quelle sera l'opération effectuée en premier, l'addition ou la multiplication ? Si c'est l'addition, le résultat sera 8 * 8, soit 64. Si c'est la multiplication, le résultat sera 5 + 24, soit 29.

La norme C++ ne laisse pas de place au hasard dans l'application des opérations. Tout opérateur a un niveau de priorité par rapport aux autres (voir l'Annexe C). La multiplication ayant une priorité supérieure à l'addition, c'est elle qui sera exécutée en premier et la valeur de l'expression sera donc égale à 29.

Lorsque deux opérateurs sont de priorité identique, les opérations s'effectuent de gauche à droite. Dans l'expression suivante :

```
x = 5 + 3 + 8 * 9 + 6 * 4;
```

La multiplication est prioritaire, de gauche à droite. D'où : 8 * 9 = 72, et 6 * 4 = 24. Ce qui revient à :

```
x = 5 + 3 + 72 + 24;
```

L'addition s'effectue ensuite de gauche à droite : 5 + 3 = 8 ; 8 + 72 = 80 ; 80 + 24 = 104.

Attention : certains opérateurs (l'affectation par exemple) sont évalués de *droite à gauche* !

Que faire si l'ordre des priorités ne vous convient pas ? Prenons un exemple :

```
totalSecondes = nbMinutesA + nbMinutesB * 60;
```

Dans cette expression, `nbMinutesB` sera multiplié par 60, puis ajouté à `nbMinutesA`. Or, on souhaite en fait ajouter les deux variables pour obtenir le nombre de minutes, puis multiplier ce résultat par 60 pour obtenir le nombre de secondes.

Les parenthèses permettent de définir l'ordre des priorités de façon adéquate. L'exemple précédent devrait donc s'écrire de la façon suivante :

```
totalSecondes = (nbMinutesA + nbMinutesB) * 60;
```

Parenthèses imbriquées

Dans les expressions complexes, vous pouvez imbriquer des parenthèses. Par exemple, pour déterminer le nombre total de secondes puis calculer le nombre total de personnes concernées avant de multiplier le nombre de secondes par le nombre de personnes, on écrira l'expression suivante :

```
totalPersonSecondes = ( ( (nbMinutesA +
                           nbMinutesB) * 60) *
                          (personTravail + personVacances) );
```

Cette expression complexe se lit de l'intérieur vers l'extérieur. Elle ajoute d'abord `nbMinutesA` à `nbMinutesB`, puis multiplie le résultat par `60`. Ensuite, elle additionne `personTravail` et `personVacances` et multiplie cette somme par le nombre de secondes.

Cet exemple met en évidence un problème important : cette expression est facile à comprendre pour un ordinateur mais très difficile à lire, modifier et comprendre pour un humain. On pourrait la décomposer en utilisant des variables entières temporaires : ,

```
totalMinutes = nbMinutesA + nbMinutesB;
totalSecondes = totalMinutes * 60;
totalPersonnel = personTravail + personVacances;
totalPersonSecondes = totalPersonnel * totalSecondes;
```

Cet exemple est plus facile à comprendre, même s'il est plus long à écrire et qu'il utilise plus de variables temporaires que le précédent. Si vous préférez ce type de programmation, n'oubliez pas d'insérer des commentaires et de remplacer 60 par une constante symbolique : vous aurez alors un code qui sera plus facile à comprendre et à maintenir.

Faire	*Ne pas faire*
● Se souvenir que les expressions ont une valeur. ● Pour incrémenter ou décrémenter une variable avant son utilisation dans l'expression, utiliser l'opérateur préfixe (++*variable*). ● Pour incrémenter ou décrémenter une variable après son utilisation dans l'expression, utiliser l'opérateur suffixe (*variable*++). ● Modifier les priorités de calcul à l'aide de parenthèses.	● Avoir recours à des imbrications trop profondes qui rendent l'expression difficile à comprendre et à mettre à jour. ● Confondre le suffixe et le préfixe.

Vrai ou faux

Toute expression peut être évaluée en terme de vrai ou faux. Une expression qui donne un résultat mathématique de zéro renverra `false`, et `true` dans tous les autres cas. ,

Dans les versions précédentes de C++, les notions de vrai ou faux étaient représentées par des entiers, mais la norme ANSI a introduit le type `bool`. Une variable de ce type peut avoir deux valeurs : `true` (vrai) ou `false` (faux).



De nombreux compilateurs proposaient déjà un type `bool`, qui était représenté en interne par un entier `long` et occupait donc 4 octets. Aujourd'hui, les compilateurs compatibles ANSI proposent souvent un type `bool` qui n'occupe qu'un seul octet.

Opérateurs relationnels

Les opérateurs relationnels permettent d'évaluer l'égalité ou la différence entre deux nombres. Une instruction relationnelle produit toujours un résultat `true` ou `false`. Les différents opérateurs disponibles sont présentés dans le Tableau 4.1.



Tous les opérateurs relationnels renvoient une valeur de type `bool` : soit `true`, soit `false`. Dans les précédentes versions du C++, ces opérateurs renvoyaient `0` (pour `false`) ou une valeur non nulle (généralement `1`) pour `true`.

Si la variable entière `monAge` est égale à 45 et que la variable entière `tonAge` est égale à 50, vous pouvez tester si elles sont égales à l'aide de l'opérateur d'égalité (==) :

```
// La valeur de monAge est-elle égale à celle de tonAge ?
monAge == tonAge;
```

La valeur renvoyée est `false` puisque les variables sont différentes. Vous pouvez vérifier que `monAge` est inférieure à `tonAge` grâce à l'expression :

```
// La valeur de moAge est-elle inférieure à celle de tonAge ?
monAge < tonAge;
```

La réponse est `true`, 45 étant inférieur à 50.



Certains programmeurs débutants confondent l'opérateur d'affectation (=) avec l'opérateur d'égalité (==). Il en résulte des bogues dans les programmes.

Il existe six opérateurs relationnels : égal à (==), inférieur à (<), supérieur à (>), inférieur ou égal à (<=), supérieur ou égal à (>=) et différent de (!=). Le Tableau 4.1 dresse la liste des opérateurs relationnels.

Tableau 4.1 : Opérateurs relationnels

Nom	*Opérateur*	*Exemple*	*Résultat*
Egal à	==	`100 == 50;` `50 == 50;`	faux vrai
Différent de	!=	`100 != 50;` `50 != 50;`	vrai faux
Supérieur à	>	`100 > 50;` `50 > 50;`	vrai faux
Supérieur ou égal à	>=	`100 >= 50;` `50 >= 50;`	vrai vrai
Inférieur à	<	`100 < 50;` `50 < 50;`	faux faux
Inférieur ou égal à	<=	`100 <= 50;` `50 <= 50;`	faux vrai

Faire	*Ne pas faire*
● Se rappeler que les opérateurs relationnels renvoient une valeur `true` (vrai) ou `false` (faux).	● Confondre l'opérateur d'affectation (=) avec l'opérateur égal à (==). C'est l'une des erreurs les plus fréquentes en C++.

L'instruction *if*

Cette instruction permet de tester une condition (si deux variables sont égales, par exemple) et de se brancher sur un bloc d'instructions ou sur un autre en fonction du résultat obtenu.

La forme la plus simple de l'instruction `if` est :

```
if (expression)
   instruction;
```

L'expression entre parenthèses peut contenir n'importe quelle expression, mais le plus souvent il s'agit d'une expression relationnelle. Si le résultat est faux, l'instruction est ignorée ; s'il est vrai, l'instruction est exécutée. Voici un exemple :

```
if (grandNombre > petitNombre)
   grandNombre = petitNombre;
```

Ce code compare `grandNombre` et `petitNombre`. Si `grandNombre` est supérieur à `petitNombre`, la ligne suivante affecte la valeur de `petitNombre` à `grandNombre`. Dans le cas contraire, la seconde ligne est ignorée.

Un bloc d'instruction entouré par des accolades étant équivalent à une instruction simple, le branchement peut effectuer plusieurs instructions :

```
if (expression)
{
   instruction1;
   instruction2;
   instruction3;
}
```

Exemple :

```
if (grandNombre > petitNombre)
{
   grandNombre = petitNombre;
   std::cout << "grandNombre : " << grandNombre << "\n";
   std::cout << "petitNombre : " << petitNombre << "\n";
}
```

Cette fois, si `grandNombre` est supérieur à `petitNombre`, la valeur de `petitNombre` sera affectée à `grandNombre` et les valeurs de ces deux variables seront affichées. Le Listing 4.4 est un exemple plus complet d'utilisation des opérateurs relationnels pour effectuer des branchements.

Listing 4.4 : Branchements conditionnels

```
1:  // Branchements conditionnels à l'aide
2:  // d'opérateurs relationnels
3:  #include <iostream>
4:  int main()
5:  {
6:     using std::cout;
7:     using std::cin;
8:
9:     int OMScore, PSGScore;
```

```
10:     cout << "Score de l'OM : ";
11:     cin >> OMScore;
12:
13:     cout << "\nScore du PSG : ";
14:     cin >> PSGScore;
15:
16:     cout << "\n";
17:
18:     if (OMScore > PSGScore)
19:        cout << "Allez l'OM !\n";
20:
21:     if (OMScore < PSGScore)
22:     {
23:        cout << "Vive le PSG !\n";
24:     }
25:
26:     if (OMScore == PSGScore)
27:     {
28:        cout << "Egalite ? Oh non !\n";
29:        cout << "Donne-moi le score du PSG : ";
30:        cin >> PSGScore;
31:
32:        if (OMScore > PSGScore)
33:           cout << "Je le savais, allez l'OM !";
34:
35:        if (PSGScore > OMScore)
36:           cout << "Je le savais, vive le PSG !";
37:
38:        if (PSGScore == OMScore)
39:           cout << "Match nul !";
40:     }
41:
42:     cout << "\nOK.\n";
43:     return 0;
44: }
```

Ce programme produit par exemple le résultat suivant :

```
Score de l'OM : 2
Score du PSG : 2
Egalite ? Oh non !
Donne-moi le score du PSG : 2
Match nul !
OK.
```

Ce programme demande les scores de deux équipes de football. Les variables sont comparées aux lignes 18, 21 et 26.

Si l'un des scores est supérieur à l'autre, un message apparaît. Si les scores sont égaux, le bloc de code de la ligne 26 à la ligne 40 s'exécute. Le second score est demandé à nouveau, puis les scores sont comparés.

Notez que si le score initial du PSG est supérieur à celui de l'OM, la condition de l'instruction `if` (ligne 18) donne `false` et la ligne 19 n'est pas exécutée. Si la ligne 21 renvoie une valeur `true`, l'instruction de la ligne 23 s'exécute. Le test de la ligne 26 s'exécute et si le résultat est faux, le programme passe directement à la ligne 41, sautant tout le bloc d'instructions.

Cet exemple montre qu'un résultat vrai à un test `if` n'empêche pas l'évaluation des autres instructions `if`.

Notez que l'action des deux premières instructions `if` tient sur une ligne (affichage de "Allez l'OM" ou "Vive le PSG"). Dans le premier exemple (ligne 19) nous n'avons pas utilisé d'accolades car elles ne sont pas obligatoires pour un bloc d'une seule instruction, mais elles peuvent être ajoutées comme aux lignes 22 à 24.

Éviter les erreurs classiques avec les instructions `if`

De nombreux programmeurs C++ débutants ajoutent par inadvertance un point-virgule après les instructions `if` :,

```
if(uneValeur < 10);    // Attention ! Remarquez le point-virgule.
   uneValeur = 10;
```

Le but ici était de tester si `uneValeur` était inférieure à 10 et, dans l'affirmative, de lui affecter la valeur `10`. Si vous exécutez le code tel qu'il est écrit ci-dessus, vous constaterez que `uneValeur` est toujours mis à `10` ! Pourquoi ? L'instruction `if` se termine par un point-virgule ce qui correspond à un opérateur vide (qui ne fait rien).

L'indentation n'ayant aucun sens pour le compilateur, le code précédent revient à écrire :

```
if (uneValeur < 10)  // test
;  // ne rien faire
uneValeur = 10;  // affectation
```

La suppression du point-virgule à la fin de l'instruction `if` résoudra le problème.

Pour réduire les risques, vous pouvez toujours écrire vos instructions `if` entre accolades, même lorsque le corps ne s'étale que sur une ligne :

```
if (uneValeur < 10)
{
   uneValeur = 10;
}
```

L'indentation

Le Listing 4.4 présente un des styles d'indentation pour les instructions `if`. Le choix du meilleur style pour l'alignement des accolades est un sujet qui déclenche toujours les passions parmi les programmeurs. Il en existe des douzaines, mais vous rencontrerez le plus souvent les trois suivants :

- L'accolade d'ouverture est placée à la suite de la condition et l'accolade de fermeture est alignée sur le `if`.

```
if (expression) {
    instructions
}
```

- Les deux accolades sont alignées sous le `if` et les instructions sont décalées.

```
if (expression)
{
    instructions
}
```

- Les accolades et les instructions sont décalées.

```
if (expression)
   {
   instructions
   }
```

Cet ouvrage utilise le deuxième style, car les blocs apparaissent plus clairement lorsque leurs accolades sont alignées avec la condition. Cela dit, vous êtes tout à fait libre d'adopter la méthode de votre choix du moment que vous restez cohérent.

L'instruction *else*

Un programme doit souvent effectuer un branchement lorsqu'une condition est vraie et un autre si elle est fausse. Cela évite les redondances dans les fichiers sources (voir le Listing 4.5).

Le mot-clé `else` permet d'effectuer ces branchements de façon claire :

```
if (expression)
   instruction;
else
   instruction;
```

Listing 4.5 : Utilisation du mot-clé *else*

```
 1:   // Utilisation de la clause else
 2:   //
 3:  #include <iostream>
 4:  int main()
 5:  {
 6:     using std::cout;
 7:     using std::cin;
 8:
 9:     int premierNombre, secondNombre;
10:     cout << "Entrez un nombre : ";
11:     cin >> premierNombre;
12:     cout << "\nEntrez un nombre plus petit : ";
13:     cin >> secondNombre;
14:     if (premierNombre > secondNombre)
15:        cout << "\nMerci !\n";
16:     else
17:        cout << "\nErreur ! Le premier n'est pas superieur !";
18:
19:     return 0;
20:  }
```

Ce programme produit le résultat suivant :

```
Entrez un nombre : 10
Entrez un nombre plus petit : 12
Erreur ! Le premier n'est pas superieur !
```

La ligne 14 évalue la condition de l'instruction `if`. Si elle est vraie, la ligne 15 s'exécute et le programme avance jusqu'à la ligne 18 (après l'instruction `else`). Si la condition de la ligne 14 renvoie `false`, le flux de contrôle passe à la clause `else` et c'est la ligne 17 qui s'exécute. Si la clause `else` était supprimée, la ligne 17 s'exécuterait à chaque fois, quel que soit le résultat de la condition de l'instruction `if`.

Les deux instructions du `if` et du `else` peuvent être remplacées par des blocs entre accolades.

Instruction *if*

La syntaxe de l'instruction `if` est la suivante :

Forme 1

```
if (expression)
   instruction;
instruction_suivante;
```

Si *expression* est vraie, l'*instruction* s'exécute et le programme continue jusqu'à l'*instruction_suivante*. Si *expression* est fausse, l'exécution passe directement à l'*instruction_suivante*.

L'*instruction* peut être une expression simple ou un bloc d'instructions entre accolades.

Forme 2

```
if (expression)
   instruction1;
else
   instruction2;
instruction_suivante;
```

Si *expression* est vraie, l'*instruction1* s'exécute. Dans le cas contraire, c'est l'*instruction2*. Le programme passe ensuite à l'*instruction_suivante*.

Exemple :

```
if (val < 10)
   cout << "val est inferieur à 10");
else
   cout << "val n'est pas inferieur à 10!");
cout << "Termine." << endl;
```

Instructions *if* imbriquées

Une instruction `if...else` peut contenir une autre instruction `if...else`. Cette pratique s'appelle l'imbrication de conditions :

```
if (expression1)
{
   if (expression2)
      instruction1;
   else
   {
      if (expression3)
         instruction2;
      else
         instruction3;
   }
}
else
   instruction4;
```

Les choses se compliquent ! Si *`expression1`* est vraie et *`expression2`* est vraie, l'*`instruction1`* s'exécute. Si *`expression1`* est vraie alors que *`expression2`* est fausse, et que *`expression3`* est vraie, l'*`instruction2`* s'exécute. Si *`expression1`* est vraie, et que

les *expressions2* et *3* sont fausses, l'*instruction3* s'exécute. Enfin, si *expression1* est fausse, l'*instruction4* s'exécute. Comme vous pouvez le constater, les instructions `if` imbriquées peuvent devenir assez confuses !

Le Listing 4.6 montre un exemple d'utilisation de cette structure.

Listing 4.6 : Conditions imbriquées

```
// Listing 4.6 - instructions if imbriquées
//
#include <iostream>
int main()
{
   // Demander deux nombres
   // Les affecter à nb1 et nb2
   // Si nb1 est supérieur ou égal à nb2,
   // vérifier s'ils sont divisibles
   // Si oui, vérifier s'ils sont identiques

   using namespace std;

   int nb1, nb2;
   cout << "Entrez deux nombres.\nPremier : ";
   cin >> nb1;
   cout << "\nSecond : ";
   cin >> nb2;
   cout << "\n\n";

   if (nb1 >= nb2)
   {
      if ( (nb1 % nb2) == 0) // sont-ils divisibles ?
      {
         if (nb1 == nb2)
            cout << "Ils sont identiques !\n";
         else
            cout << "Ils sont divisibles !\n";
      }
      else
         cout << "Ils ne sont pas divisibles !\n";
   }
   else
      cout << "Le second nombre est supérieur !\n";
   return 0;
}
```

Ce programme produit par exemple le résultat suivant :

```
Entrez deux nombres.
Premier : 15

Second : 4

Ils ne sont pas divisibles !
```

Les deux nombres sont saisis et comparés. L'instruction `if` de la ligne 21 vérifie que le premier est supérieur ou égal au second. Si ce n'est pas le cas, la clause `else` de la ligne 33 s'exécute.

Si le premier `if` est vrai, la ligne 22 s'exécute et le second `if` est testé. Il vérifie que la division ne produit pas de reste, ce qui signifie que les nombres sont divisibles ou égaux. L'instruction `if` de la ligne 25 vérifie s'ils sont égaux et affiche un message approprié .

Si l'instruction `if` de la ligne 23 échoue, c'est la clause `else` de la ligne 30 qui s'exécute.

Utilisation d'accolades dans les instructions *if* imbriquées

Bien qu'il soit permis de ne pas mettre d'accolades pour des instructions `if` composées d'une seule instruction et que l'on puisse ainsi imbriquer des instructions `if`, cela peut entraîner une grosse confusion. Le code suivant, par exemple, est parfaitement autorisé en C++ mais semble bien confus :

```
if (x > y)              // si x > y
   if (x < z)           // et x < z
      x = y;            // x reçoit la valeur de y
   else                 // sinon, si x n'est pas inferieur à z
      x = z;            // definir x sur la valeur de z
else                    // sinon, si x n'est pas superieur à y
   y = x;               // definir y sur la valeur de x
```

Les indentations et les espaces sont pratiques pour le programmeur, mais ne changent rien pour le compilateur. Il est très facile de se tromper et d'affecter une clause `else` au mauvais `if`, comme le montre le Listing 4.7

Listing 4.7 : Intérêt des accolades pour clarifier les instructions conditionnelles imbriquées

```
1:   // Importance des accolades
2:   // dans des instructions if imbriquées
3:  #include <iostream>
```

```
 4:  int main()
 5:  {
 6:     int x;
 7:     std::cout << "Entrez un nombre inférieur à 10 ou supérieur à 100 : ";
 8:     std::cin >> x;
 9:     std::cout << "\n";
10:
11:     if (x >= 10)
12:        if (x > 100)
13:           std::cout << "Supérieur à 100, Merci !\n";
14:     else                        // pas la directive else prévue !
15:        std::cout << "Inférieur à 10, Merci !\n";
16:
17:     return 0;
18:  }
```

Ce programme produit par exemple le résultat suivant :

```
Entrez un nombre inférieur à 10 ou supérieur à 100 : 30
Inférieur à 10, Merci !
```

L'intention du programmeur, ici, était de vérifier que le nombre saisi était inférieur à 10 ou supérieur à 100 et d'envoyer un message de remerciement.

Lorsque la condition de l'instruction `if` de la ligne 11 est vraie, la ligne 12 s'exécute. Ici, la ligne 12 s'exécute si le nombre entré est supérieur ou égal à 10. La ligne 12 contient aussi une instruction `if` dont la condition est vraie si le nombre entré est supérieur à 100. Dans ce cas, l'instruction de la ligne 13 s'exécute et affiche le message qui convient.

Si le nombre entré est inférieur à 10, la condition de la ligne 11 renvoie `false`. Le programme se poursuit alors à la ligne suivant cette instruction `if` (ici, la ligne 16). Si vous entrez un nombre inférieur à 10, vous obtiendrez donc :

```
Entrez un nombre inferieur a 10 ou superieur a 100 : 9
```

Comme vous le constatez, aucun message ne s'affiche. La clause `else` de la ligne 14 était censée correspondre à l'instruction `if` de la ligne 11, comme l'indiquait son indentation. Malheureusement, elle est en fait associée à la clause `if` de la ligne 12, ce qui entraîne le problème que nous venons de constater.

Ce bogue est subtil car le compilateur ne le signale pas ! C'est un programme syntaxiquement correct, mais qui ne fait tout simplement pas ce que l'on attend de lui. En outre, il fonctionne correctement la plupart du temps, tant que l'on entre un nombre supérieur à 100. Cependant, avec un nombre compris entre 11 et 99, vous constaterez qu'il y a un problème évident !

Le Listing 4.8 corrige ce problème en utilisant des accolades.

Listing 4.8 : Utilisation correcte d'accolades dans une instruction *if*

```
 1:    // Utilisation correcte d'accolades
 2:    // dans les instructions if imbriquées
 3:  #include <iostream>
 4:  int main()
 5:  {
 6:     int x;
 7:     std::cout << "Entrez un nombre inférieur à 10 ou supérieur à 100 : ";
 8:     std::cin >> x;
 9:     std::cout << "\n";
10:
11:     if (x >= 10)
12:     {
13:        if (x > 100)
14:           std::cout << "Supérieur à 100, Merci !\n";
15:     }
16:     else                              // Bogue corrigé !
17:        std::cout << "Inférieur à 10, Merci !\n";
18:     return 0;
19:  }
```

Ce programme réagit maintenant correctement, par exemple :

```
Entrez un nombre inférieur à 10 ou supérieur à 100 : 34

Entrez un nombre inférieur à 10 ou supérieur à 100 : 8
Inférieur à 10, Merci !

Entrez un nombre inférieur à 10 ou supérieur à 100 : 101
Supérieur à 100, Merci !
```

Les accolades des lignes 12 et 15 regroupent ce qu'elles contiennent en une seule instruction. La ligne 16 correspond maintenant à l'instruction `if` de la ligne 11.



Vous pouvez éviter bon nombre de problèmes liés aux instructions `if…else` *en plaçant toujours des accolades pour les instructions des clauses* `if` *et* `else`*, même lorsque la condition n'est suivie que d'une seule instruction :*

```
if (valeur < 10)
{
   valeur = 10;
}
else
{
   valeur = 25;
};
```

Opérateurs logiques

On veut parfois tester plusieurs conditions à la fois, par exemple pour savoir si "x est supérieur à y et si y est supérieur à z". Le programme doit déterminer si ces deux conditions sont vérifiées ou non afin d'agir en conséquence.

Imaginons un système d'alarme évolué. Si l'alarme de la porte sonne ET s'il est plus de 18 h ET si nous ne sommes PAS en vacances OU si c'est un week-end, appeler la police. Pour ce type d'évaluation, il faut utiliser les trois opérateurs logiques de C++, qui sont présentés dans le Tableau 4.2.

Tableau 4.2 : Opérateurs logiques

Opérateur	*Symbole*	*Exemple*
`AND`	`&&` (et)	`expression1 && expression2`
`OR`	`||` (ou)	`expression1 || expression2`
`NOT`	`!` (non)	`!expression`

ET logique

Il permet d'évaluer deux expressions. Pour renvoyer un résultat vrai, les deux expressions doivent être toutes les deux vraies. S'il est vrai que vous avez faim ET que vous ayez de l'argent, ALORS vous pouvez acheter une pizza. Par conséquent,

```
if ( (x == 5) && (y == 5) )
```

sera vérifiée si `x` et `y` sont tous les deux égaux à 5.

Notez que l'opérateur logique `ET` est symbolisé par deux symboles `&&`, ce qui est différent de l'opérateur binaire `&` présenté au Chapitre 21.

OU logique

Il permet d'évaluer deux expressions. Si l'une ou l'autre est vraie, le résultat est vrai. Si vous avez de l'argent liquide OU une carte de crédit (ou les deux), ALORS vous pouvez payer l'addition. Par conséquent,

```
if ( (x == 5) || (y == 5) )
```

sera vérifiée si `x` ou `y` est égal à 5, ou si tous les deux sont égaux à 5.

L'opérateur logique `OU` est symbolisé par deux barres verticales (`||`). La barre verticale simple symbolise un opérateur différent qui sera étudié au Chapitre 21.

NON logique

Il donne un résultat vrai si l'expression testée est fausse. Par exemple, l'expression :

```
if ( !(x == 5) )
```

ne sera vraie que si x n'est pas égal à 5, ce qui revient à écrire :

```
if (x != 5)
```

Évaluation en court-circuit

Lors de l'évaluation d'une instruction ET comme :

```
if ( (x == 5) && (y == 5) )
```

le compilateur teste d'abord la véracité de la première instruction (x==5) ; en cas d'échec (si x n'est pas égal à 5), la seconde instruction (y == 5) ne sera pas testée puisque la clause ET requiert que les deux conditions soient vraies.

De même, dans le cas d'une clause OR comme :

```
if ( (x == 5) || (y == 5) )
```

si la première instruction donne vrai (x == 5), la seconde ne sera pas exécutée (y == 5) puisque l'une ou l'autre des conditions est suffisante.

Bien que cela puisse ne pas sembler important, étudiez l'exemple suivant :

```
if ( (x == 5 )|| (++y == 3) )
```

Si x n'est pas égal à 5, (++y == 3) ne sera pas évalué. Si vous comptiez sur l'incrémentation de y, celle-ci risque de ne pas toujours être exécutée.

Priorité des opérateurs relationnels

Comme pour toutes les expressions C++, l'utilisation des opérateurs relationnels et des opérateurs logiques, renvoie une valeur qui est, ici, true ou false. Pour connaître l'ordre dans lequel ils sont exécutés, reportez-vous à l'Annexe C. Voici un exemple :

```
if ( x > 5 &&  y > 5  || z > 5)
```

On peut supposer que le programmeur souhaite que l'expression soit vraie si x et y sont supérieurs à 5 ou si z est supérieur à 5. Cela peut aussi signifier que l'expression n'est vraie que si x est supérieur à 5, et si soit y soit z est supérieur à 5.

Si x est égal à 3 et que y et z valent tous les deux 10, la première interprétation sera vraie (z est supérieur à 5, on ignore donc x et y), mais la seconde sera fausse (il n'est pas vrai que x soit supérieur à 5 et peu importe ce qui se trouve du côté droit du symbole && puisque les deux côtés doivent être vrais).

Les parenthèses permettent donc d'affecter explicitement des priorités et rendent ainsi l'expression plus lisible :

```
if (  (x > 5)  && (y > 5 ||  z > 5) )
```

Cette instruction donne un résultat faux, car x n'est pas supérieur à 5. La partie gauche de l'expression AND étant fausse, toute l'expression est fausse.



Pour clarifier vos intentions, n'hésitez pas à insérer des parenthèses. Elles précisent ce que vous voulez faire et évitent les erreurs nées d'une mauvaise compréhension de l'ordre des opérateurs.

Vrai ou faux

En C++, la valeur zéro équivaut à `false` alors que les autres valeurs sont considérées comme `true`. Une expression ayant toujours une valeur, de nombreux programmeurs C++ exploitent cette fonctionnalités dans leurs programmes. Une instruction comme :

```
if (x)           // si x est vrai (non zéro)
   x = 0;
```

peut ainsi se lire "si x est différent de zéro, x reçoit la valeur 0". Ce qui revient à écrire :

```
if (x != 0)      // si x est différent de zéro
   x = 0;
```

Ces deux instructions sont correctes, mais la seconde est plus lisible. Il est donc conseillé de réserver la première forme aux véritables tests logiques plutôt que de s'en servir pour tester si une valeur est nulle.

Ces deux instructions sont également équivalentes :

```
if (!x)           // si x est faux (zéro)
if (x == 0)       // si x égale zéro
```

La seconde instruction est plus explicite si vous testez la valeur mathématique de x plutôt que son état logique.

Faire	*Ne pas faire*
● Mettre des parenthèses dans vos tests logiques pour les rendre plus clairs et définir explicitement les priorités. ● Utiliser des accolades pour clarifier les instructions `if` imbriquées.	● Utiliser `if (x)` comme synonyme de `if (x != 0);`. ● Utiliser `if (!x)` comme synonyme de `if (x == 0);`.

Opérateur conditionnel (ternaire)

L'opérateur conditionnel (`?:`) est le seul opérateur C++ ternaire (il comprend trois termes).

Il utilise trois expressions et renvoie une valeur :

```
(expression1) ? (expression2) : (expression3)
```

Cette ligne signifie : "Si *expression1* est vraie, renvoyer la valeur de *l'expression2* ; sinon, renvoyer la valeur de *l'expression3*". Le résultat de cet opérateur est généralement affecté à une variable. Le Listing 4.9 montre comment l'utiliser à la place d'une instruction `if`.

Listing 4.9 : Exemple d'opérateur conditionnel

```
 1:   // Listing 4.9 -
 2:   //
 3:  #include <iostream>
 4:  int main()
 5:  {
 6:     using namespace std;
 7:
 8:     int x, y, z;
 9:     cout << "Entrez deux nombres.\n";
10:     cout << "Premier : ";
11:     cin >> x;
12:     cout << "\nSecond : ";
13:     cin >> y;
14:     cout << "\n";
15:
16:     if (x > y)
17:        z = x;
18:     else
19:        z = y;
20:
```

```
21:     cout << "Après le test if, z: " << z;
22:     cout << "\n";
23:
24:     z =  (x > y) ? x : y;
25:
26:     cout << "Après le test conditionnel, z: " << z;
27:     cout << "\n";
28:     return 0;
29: }
```

Ce programme produit le résultat suivant :

```
Entrez deux nombres.
Premier : 5

Second : 8

Après le test if, z: 8
Après le test conditionnel, z: 8
```

Trois variables entières sont créées : `x`, `y` et `z`. Les deux premières sont saisies par l'utilisateur. L'instruction `if` de la ligne 16 détermine la plus grande et l'affecte à `z`. Le résultat s'affiche à la ligne 21.

L'opérateur conditionnel de la ligne 24 réalise le même test et affecte la valeur la plus élevée à `z`. Si `x` est supérieur à `y`, il renvoie la valeur de `x` ; sinon, il renvoie la valeur de `y`. Cette valeur est affectée à `z` et le résultat s'affiche à la ligne 26. Comme vous pouvez le constater, l'instruction conditionnelle est en fait un équivalent (raccourci) de l'instruction `if...else`.

Questions-réponses

Q Pourquoi utiliser des parenthèses superflues lorsque la priorité des opérateurs est évidente ?

R Parce que le programme gagne en lisibilité et peut être mis à jour plus aisément.

Q Si les opérateurs relationnels renvoient `true` ou `false`, pourquoi une valeur différente de zéro est-elle considérée comme vraie ?

R Cette convention provient du langage C, qui était souvent utilisé pour écrire les logiciels de bas niveau, comme les systèmes d'exploitation et les logiciels de contrôle en temps réel. Il est probable que cette utilisation ait évolué pour devenir un raccourci permettant de tester rapidement si tous les bits d'un masque ou d'une variable sont égaux à 0.

Les opérateurs relationnels renvoient `true` ou `false`, mais chaque expression renvoie une valeur et ces valeurs peuvent être évaluées dans une instruction `if`. Voici un exemple :

```
if ( (x = a + b) == 35 )
```

Cette instruction est tout à fait correcte en C++. Elle produit une valeur, même si `a + b` ne donne pas 35. Notez aussi que la somme obtenue est affectée à `x` dans tous les cas.

Q À quoi servent les tabulations, les espaces et les sauts de ligne dans un programme ?

R Ils n'ont aucun effet sur le déroulement du programme, mais rendent le code plus lisible.

Q Les nombres négatifs peuvent-ils posséder les valeurs `true` ou `false` ?

R Tous les nombres différents de zéro, qu'ils soient positifs ou négatifs, sont considérés comme vrais.

Testez vos connaissances

1. Qu'est-ce qu'une expression ?
2. Est-ce que `x = 5 + 7` est une expression ? Quelle est sa valeur ?
3. Quelle est la valeur de `201 / 4` ?
4. Quelle est la valeur de `201 % 4` ?
5. Si `monAge`, `a` et `b` sont des variables entières, quel est le résultat de ces expressions ?

```
monAge = 39;
a = monAge++;
b = ++monAge;
```

6. Quelle est la valeur de `8 + 2*3` ?
7. Quelle est la différence entre `if (x = 3)` et `if (x == 3)` ?
8. Les valeurs suivantes sont-elles égales à `true` ou à `false` ?

   a. `0`

   b. `1`

   c. `-1`

   d. `x = 0`

   e. `x == 0   // en admettant que x ait la valeur 0`

Exercices

1. Écrivez une instruction `if` qui teste deux variables entières et qui affecte la valeur la plus faible à la variable la plus élevée. N'utilisez qu'une seule clause `else`.
2. Lisez ce programme. Imaginez la saisie de trois nombres et écrivez le résultat.

```
1:   #include <iostream>
2:   using namespace std;
3:   int main()
4:   {
5:        int a, b, c;
6:        cout << "Entrez trois nombres.\n";
7:        cout << "a : ";
8:        cin >> a;
9:        cout << "\nb : ";
10:        cin >> b;
11:       cout << "\nc : ";
12:       cin >> c;
13:
14:       if (c = (a-b))
15:            cout << "a :" << a << " moins b :" << b <<
16:                         " est égal à c :" << c << "\n";
17:       else
18:            cout << "a-b n'est pas égal à  c :";
19:    return 0;
20:  }
```

3. Saisissez le programme de l'Exercice 2 ; compilez-le, liez-le et lancez-le. Entrez les nombres 10, 20 et 50. Obtenez-vous le résultat attendu ? Pourquoi ?
4. Examinez le programme suivant et imaginez le résultat :

```
1:     #include <iostream>
2:     using namespace std;
3:      int main()
4:      {
5:           int a = 2, b = 2, c;
6:           if (c = (a-b))
7:                cout << "c vaut : " << c;
8:     return 0;
9:      }
```

5. Saisissez, compilez et lancez le programme précédent. Obtenez-vous le résultat attendu ? Pourquoi ?



5

Fonctions

Au sommaire de ce chapitre

- Le rôle d'une fonction et ses différents éléments
- Comment déclarer et définir les fonctions
- Comment passer des paramètres à une fonction
- Comment renvoyer une valeur à partir d'une fonction

On pourrait penser que la programmation orientée objet a privilégié les objets au détriment des fonctions. Il n'en est rien : ces dernières demeurent un élément essentiel des applications. Les fonctions globales existent en dehors des objets et des classes, tandis que les fonctions membres (ou méthodes membres) existent au sein d'une classe et accomplissent ses différentes tâches.

Nous ne présenterons ici que les fonctions globales ; nous étudierons le principe des fonctions membres au cours du prochain chapitre.

Qu'est-ce qu'une fonction ?

Par nature, une fonction est un sous-programme qui agit sur des données et renvoie une valeur. Tout programme C++ comprend au moins une fonction : la fonction `main()`, qui est la fonction principale qui s'exécute automatiquement et qui peut appeler d'autres fonctions, qui peuvent elles-mêmes en appeler d'autres.

Ces fonctions ne faisant pas partie d'un objet, elles sont appelées "globales", c'est-à-dire qu'elles sont accessibles de n'importe quel endroit du programme. Les fonctions abordées dans ce chapitre sont les fonctions globales, sauf mention contraire.

Chaque fonction porte un nom particulier qui l'identifie dans le programme. Lorsque ce nom est rencontré, le programme se place directement au début de la fonction – c'est que l'on désigne par le terme d'*appel* de fonction. Lorsque la fonction se termine (en rencontrant une instruction `return` ou l'accolade fermante de la fonction), le programme poursuit son exécution à la ligne qui suit l'appel de la fonction (voir Figure 5.1).

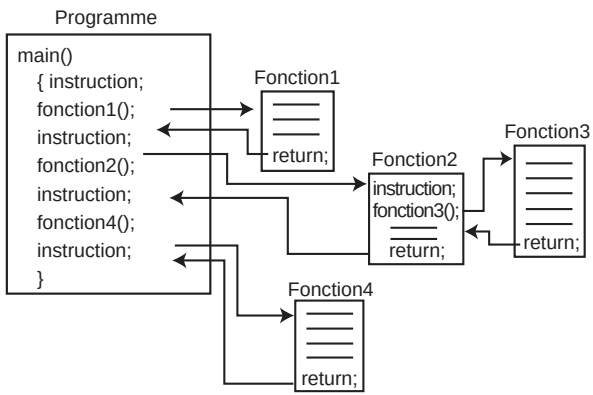


Figure 5.1
Après l'exécution d'une fonction, le programme revient à l'instruction qui suit l'appel.

Lorsqu'elle est bien conçue, une fonction réalise une seule action spécifique, identifiée par le nom de la fonction. Les tâches complexes doivent être divisées en fonctions simples qui seront appelées tour à tour.

Il existe deux catégories de fonctions : les fonctions définies par l'utilisateur et les fonctions prédéfinies qui sont intégrées au compilateur. Les fonctions définies par l'utilisateur sont celles que vous écrivez vous-même.

Valeurs renvoyées, paramètres formels et effectifs

Vous l'avez vu au Chapitre 2, les fonctions peuvent recevoir des valeurs et *renvoyer* une valeur. Lorsqu'elle est appelée, une fonction effectue une certaine tâche puis renvoie un résultat à l'appelant. Cette valeur s'appelle la *valeur renvoyée* et son type doit être déclaré. Lorsque vous écrivez :

```
int maFonction();
```

vous déclarez donc que `maFonction` renverra une valeur entière. Étudiez maintenant la déclaration suivante :

```
int maFonction(int unEntier, float unFlottant);
```

Cette déclaration indique que `maFonction` renverra une valeur entière et prendra deux valeurs.

Lorsque vous envoyez des valeurs *à* une fonction, celles-ci agissent comme des variables que vous pouvez manipuler à l'intérieur de la fonction. La description des valeurs envoyées est appelée *liste des paramètres formels*. Dans l'exemple précédent, cette liste contient `unEntier`, qui est une variable entière et `unFlottant`, une variable de type `float`.

Un paramètre formel décrit le *type* de la valeur passée à la fonction. La valeur qui sera réellement passée lors de l'appel est appelée *paramètre effectif* (ou *réel*). Dans l'exemple suivant :

```
int valeurRenvoyee = maFonction(5, 6.7);
```

la variable entière `valeurRenvoyee` reçoit la valeur renvoyée par `maFonction` et les valeurs `5` et `6.7` sont passées en paramètres. Le type des paramètres effectifs doit correspondre au type des paramètres formels correspondants. Dans ce cas, `5` va dans un entier et `6.7` dans une variable `float` ; les types correspondent donc bien.

Déclaration et définition de fonctions

Pour être exploitable dans le programme, une fonction doit être déclarée puis définie. La déclaration indique au compilateur le nom de la fonction, le type de la valeur qu'elle renvoie et les paramètres attendus par la fonction. La définition décrit au compilateur le fonctionnement de la fonction.

Il est impossible d'appeler une fonction qui n'a pas au préalable été déclarée. La déclaration d'une fonction est désignée sous le nom de *prototype*.

Il existe trois manières de déclarer une fonction. Vous pouvez :

- écrire le prototype dans un fichier, que vous incluerez dans votre fichier source à l'aide de la directive `#include` ;
- écrire le prototype de la fonction dans le fichier source du programme qui l'utilise ;
- définir la fonction avant qu'elle ne soit appelée par une autre fonction. Dans ce cas, la définition tient lieu de prototype.

Bien que l'on puisse définir une fonction avant de l'utiliser, ce qui épargne de créer un prototype, cette pratique n'est pas conseillée pour trois raisons.

Tout d'abord, c'est une mauvaise idée d'obliger les fonctions à apparaître dans le fichier selon un ordre prédéterminé, car cela complique la maintenance du programme lorsque ces exigences évoluent.

Deuxièmement, il est possible qu'une fonction `A()` doive pouvoir appeler une fonction `B()` qui, elle-même, a besoin d'appeler la fonction `A()`. Il est donc impossible de définir `A()` avant de définir `B()` et de définir `B()` avant `A()`. En ce cas, il faut au moins déclarer l'une d'entre elles.

Enfin, les prototypes de fonctions facilitent la mise au point du programme. Si le prototype déclare les paramètres de la fonction et le type du résultat d'une fonction et que la fonction ne correspond pas à ce prototype, le compilateur peut détecter cette incompatibilité et produire un message, ce qui évitera que l'erreur ne soit visible qu'au moment de l'exécution. Le prototype et la définition se contrôlent mutuellement, ce qui réduit la probabilité qu'une faute de frappe provoque un bogue dans la programme.

Malgré tout, la grande majorité des programmeurs optent pour le troisième choix parce qu'il réduit le nombre de lignes de code, qu'il simplifie la maintenance (les changements d'en-tête de la fonction exigent aussi le changement du prototype) et parce que l'ordre des fonctions dans un fichier est généralement assez stable. Cela dit, les prototypes sont nécessaires dans certains cas.

Prototypes de fonctions

Les prototypes de la plupart des fonctions prédéfinies sont déjà écrits et sont stockés dans des fichiers que vous pouvez inclure dans vos programmes à l'aide de la directive `#include`. Vous devrez en revanche fournir un prototype pour les fonctions que vous écrivez vous-même.

Le prototype d'une fonction est une instruction, ce qui signifie qu'il doit se terminer par un point-virgule. Il est formé du type de la valeur renvoyée et de la *signature*. La signature d'une fonction est composée de son nom et de la liste de ses paramètres formels.

La liste des paramètres formels énumère tous les paramètres attendus et leurs types respectifs, en les séparant par des virgules, comme dans la Figure 5.2.

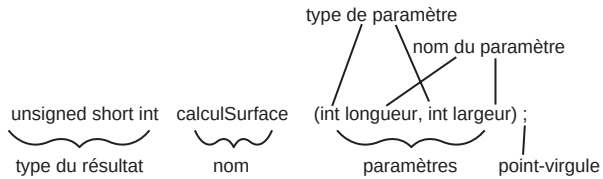


Figure 5.2
Éléments d'un prototype de fonction.

Le prototype de la fonction et sa définition doivent se correspondre, c'est-à-dire contenir les mêmes informations (type de la valeur renvoyée et signature). Si ce n'est pas le cas, cela provoque une erreur de compilation. Notez cependant que le prototype de la fonction ne comprend pas nécessairement le nom des paramètres formels, mais qu'il peut se contenter de leur type, comme dans cet exemple :

```
long Surface(int, int);
```

Ce prototype déclare la fonction `Surface()` qui renvoie un entier `long` et attend deux paramètres entiers. Bien que cette syntaxe soit correcte, il est préférable de mentionner le nom des deux paramètres car cela produit une déclaration plus compréhensible :

```
long Surface(int longueur, int largeur);
```

Le but de la fonction et de ses deux paramètres apparaît plus clairement.

Toutes les fonctions n'ont pas de type explicite pour leur résultat. En l'absence de ce type, C++ supposera que la fonction renvoit une valeur `int` (entier), mais il est conseillé de l'indiquer explicitement, y compris pour la fonction `main()`.

Si votre fonction ne renvoie pas de valeur, son type de résultat doit être `void` (vide), comme dans cet exemple :

```
void afficheNombre(int unNombre);
```

Cette ligne déclare une fonction appelée `afficheNombre` possédant un paramètre entier. `void` étant utilisé comme type de résultat, aucun élément n'est renvoyé.

Définir la fonction

La définition d'une fonction est constituée de son *en-tête* et de son *corps*. L'en-tête ressemble au prototype de la fonction, sauf que les paramètres doivent être nommés et qu'il n'y a pas de point-virgule final.

Le corps de la fonction est un ensemble d'instructions inclus entre accolades, comme dans la Figure 5.3.

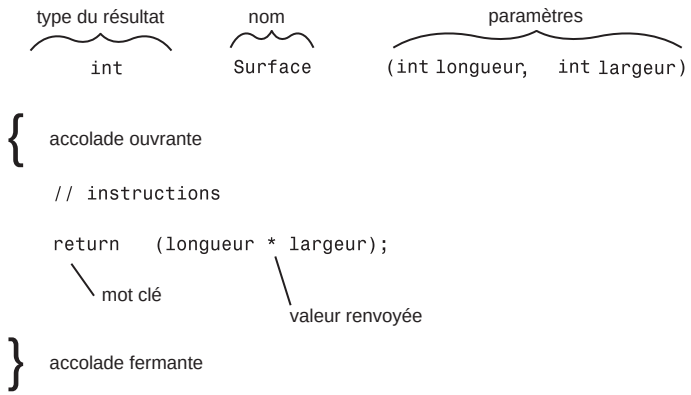


Figure 5.3
L'en-tête et le corps d'une fonction.

Le fichier source du Listing 5.1 contient un prototype déclarant une fonction dont le rôle est de calculer une surface.

Listing 5.1 : Déclaration, définition et utilisation de la fonction *Surface()*

```
// Listing 5.1 - Utilisation des prototypes de fonctions

#include <iostream>
int Surface(int longueur, int largeur); // prototype

int main()
{
   using std::cout;
   using std::cin;

   int longueurCour;
   int largeurCour;
   int surfaceCour;

   cout << "\nLargeur de votre cour ? ";
   cin >> largeurCour;
   cout << "\nLongueur de votre cour ? ";
   cin >> longueurCour;

   surfaceCour= Surface(longueurCour,largeurCour);

   cout << "\nLa surface de votre cour est de ";
   cout << surfaceCour;
```

```
24:     cout << " mètres carrés\n\n";
25:     return 0;
26:  }
27:
28:  int Surface(int longue, int large)
29:  {
30:     return longue * large;
31:  }
```

Ce programme produit le résultat suivant :

```
Largeur de votre cour ? 10

Longueur de votre cour ? 200

La surface de votre cour est de 2000 mètres carrés
```

Le prototype de la fonction `Surface()` se trouve à la ligne 4. Comparez-le à la définition de la fonction à la ligne 28. Vous pouvez constater que le nom, le type de la valeur renvoyée et les types de paramètres sont identiques. S'il y avait une différence, le compilateur produirait un message d'erreur et ne créerait pas de fichier objet. En fait, la seule différence est que le prototype se termine par un point-virgule et n'a pas de corps.

Au niveau du prototype, les paramètres s'appellent `longueur` et `largeur` et correspondent aux paramètres `longue` et `large` de la définition. Les premiers figurent dans le fichier à titre d'information alors que les seconds sont traités par la fonction. Il est souhaitable (mais non obligatoire) que les noms des paramètres dans le prototype correspondent à ceux de l'implémentation.

Les paramètres effectifs sont passés à la fonction dans l'ordre exact où ils ont été déclarés et définis, mais les noms ne sont pas pris en compte : vous auriez passé `largeurCour` suivi de `longueurCour`, la fonction aurait utilisé le premier comme longueur et la second comme largeur.



Le corps de la fonction doit toujours être placé entre accolades, même s'il n'y a qu'une seule instruction.

Exécution des fonctions

Lorsque vous appelez une fonction, son exécution commence avec l'instruction située immédiatement après l'accolade ouvrante (`{`). Cette exécution peut bien sûr contenir des parties conditionnelles (les instructions `if` et associées seront présentées plus en détail au

Chapitre 7). Les fonctions peuvent également appeler d'autres fonctions et s'appeler elles-mêmes (on parle alors de *récursivité*).

À la fin de l'exécution d'une fonction, la fonction appelante reprend la main. Pour la fonction `main()`, c'est le système d'exploitation qui reprend la main.

Portée des variables

Une variable a une *portée*, qui détermine sa durée de vie dans le programme et les endroits qui y ont accès. La portée des variables déclarées dans un bloc se limite à celui-ci ; elles ne peuvent être lues que dans ce bloc et disparaissent une fois qu'il a été exécuté. En revanche, les variables globales ont une portée globale, c'est-à-dire qu'elles sont disponibles depuis n'importe quel point du programme.

Variables locales

Non seulement on peut passer des variables à une fonction, mais on peut également déclarer des variables dans le corps de celle-ci. Les variables déclarées dans la fonction sont dites "locales", car elles n'existent que dans le cadre de cette fonction. Lorsque la fonction rend la main au programme, elles disparaissent.

Les variables locales sont des variables classiques et sont donc définies comme toutes les autres variables. Les paramètres passés à la fonction sont également considérés comme des variables locales et peuvent être utilisés comme s'ils avaient été définis dans le corps de la fonction. Le Listing 5.2 montre l'utilisation des paramètres et des variables locales dans une fonction.

Listing 5.2 : Utilisation des paramètres et des variables locales

```
 1:  #include <iostream>
 2:
 3:  float Convert(float);
 4:  int main()
 5:  {
 6:     using namespace std;
 7:
 8:     float tempFar;
 9:     float tempCel;
10:
11:     cout << "Entrez la température en degrés Fahrenheit: ";
12:     cin >> tempFar;
13:     tempCel = Convert(tempFar);
14:     cout << "\nTempérature en degrés Celsius : ";
```

```
15:     cout << tempCel << endl;
16:     return 0;
17:  }
18:
19:  float Convert(float tempFar)
20:  {
21:     float tempCel;
22:     tempCel = ((tempFar - 32) * 5) / 9;
23:     return tempCel;
24:  }
```

Quelques exemples de résultats :

```
Entrez la température en degrés Fahrenheit : 212
Température en degrés Celsius : 100

Entrez la température en degrés Fahrenheit : 32
Température en degrés Celsius : 0

Entrez la température en degrés Fahrenheit : 85
Température en degrés Celsius : 29.4444
```

Aux lignes 8 et 9, les deux variables de type `float` déclarées doivent recevoir respectivement la température en degrés Fahrenheit et en degrés Celsius. À la ligne 11, l'utilisateur entre la valeur en degrés Fahrenheit à convertir et celle-ci est passée à la fonction `Convert()` à la ligne 13.

Avec l'appel à `Convert()` ligne 13, le programme va directement à la ligne 21, c'est-à-dire à la première ligne de la fonction de conversion. La variable locale `tempCel` est alors déclarée. Attention : bien qu'elle porte le même nom, cette variable est différente de la variable `tempCel` déclarée à la ligne 9. Le programme ne risque pas de confondre les deux variables puisque la variable `tempCel` locale disparaît après l'exécution de la fonction de conversion. Le paramètre `tempFar` est une copie locale de la variable passée par la fonction `main()`.

Le paramètre formel et la variable locale pourraient s'appeler `farTemp` et `celTemp` (ou autrement), le programme fonctionnerait de la même façon. Vous pouvez essayer d'entrer d'autres noms et recompiler le programme !

La variable locale `tempCel` reçoit la valeur correspondant à la température en degrés Celsius. Il s'agit d'ôter 32 à la température en degrés Fahrenheit, de multiplier le résultat par 5 puis de le diviser par 9. Cette valeur renvoyée est affectée à la variable `tempCel` dans la fonction `main()` à la ligne 13 et affichée à la ligne 15.

Le résultat précédent montre que ce programme a été exécuté trois fois. La première fois, il montre que 212 degrés Fahrenheit correspondent au point d'ébullition de l'eau, c'est-à-dire à

100 degrés Celsius. La deuxième fois, vous pouvez constater que 32 degrés Fahrenheit correspondent au point de gel de l'eau. Enfin, le troisième test passe une valeur quelconque, qui produit un résultat fractionnaire.

Variables locales dans des blocs

Vous pouvez définir des variables à tout endroit d'une fonction, pas seulement au début. Lorsqu'elle est définie dans un bloc d'instructions, la portée d'une variable locale est limitée à ce bloc. Pour illustrer ces propos, nous vous proposons le listing suivant :

Listing 5.3 : Variables visibles au niveau du bloc

```
 1:  // Listing 5.3 - démonstration de la visibilité de
 2:  // variables au niveau du bloc
 3:
 4:  #include <iostream>
 5:
 6:  void maFct();
 7:
 8:  int main()
 9:  {
10:     int x = 5;
11:     std::cout << "\nDans main(), x vaut : " << x;
12:
13:     maFct();
14:
15:     std::cout << "\nDe retour dans main, x vaut : " << x;
16:     return 0;
17:  }
18:
19:  void maFct()
20:  {
21:     int x = 8;
22:     std::cout << "\nDans maFct, variable locale x : "
22a:              << x << std::endl;
23:
24:     {
25:        std::cout << "\nDans le bloc de maFct, x vaut : " << x;
26:
27:        int x = 9;
28:
29:        std::cout << "\nVariable x très locale : " << x;
30:     }
31:
32:     std::cout << "\nHors du bloc, dans maFct, x : "
32a:              << x << std::endl;
33:  }
```

Ce programme produit le résultat suivant :

```
Dans main(), x vaut : 5
Dans maFct, variable locale x  : 8

Dans le bloc de maFct, x vaut : 8
Variable x très locale : 9
Hors du bloc, dans maFct, x : 8

De retour dans main, x vaut : 5
```

Ce programme commence à la ligne 10 avec l'initialisation d'une variable locale `x` dans `main()`. À la ligne 11, sa valeur (5) s'affiche à l'écran.

La ligne 13 appelle la fonction `maFct()`. À la ligne 21, dans `maFct()`, une variable locale `x` est initialisée à 8. Ce chiffre s'affiche à l'écran.

L'accolade ouvrante de la ligne 24 commence un bloc d'instructions. Le contenu de la variable `x` est affiché. Une nouvelle variable également appelée `x`, mais dont la portée est limitée au bloc, est initialisée à 9 dans la ligne 27 ; c'est cette variable qui est affichée par la ligne 30.

Le bloc local se termine à la ligne 30 ; la variable locale déclarée à la ligne 27 devient *hors de portée* est n'est donc plus visible.

La valeur de la variable `x` affichée à la ligne 32 est celle qui a été déclarée à la ligne 21 dans `maFct()` ; elle est toujours égale à 8. En effet, elle n'a pas été affectée par la définition de la ligne 27 dans le bloc, puisque les portées des deux variables `x` étaient différentes.

À la ligne 33, `maFct()` devient hors de porté et sa variable locale `x` disparaît en même temps. Le programme reprend la main à la ligne 14. La ligne 15 reprend la valeur de la variable `x` créée à la ligne 10. Vous constaterez qu'elle n'a pas été modifiée par les autres affectations des variables définies dans `maFct()`.

Bien entendu, ce programme n'a qu'une valeur d'exemple et serait plus lisible si toutes les variables locales portaient des noms distincts !

Les paramètres sont des variables locales

Les paramètres passés à une fonction sont locaux à cette fonction. Les modifications qui leur sont apportées n'affectent pas les valeurs dans la fonction appelante. C'est ce que l'on appelle un passage *par valeur*, qui revient à créer une copie locale de chaque paramètre dans la fonction. Ces copies locales sont considérées comme n'importe quelle autre variable locale. Un exemple est proposé dans le Listing 5.4.

Listing 5.4 : Passage par valeur

```
 1:  // Listing 5.4 - Démonstration de passage PAR VALEUR
 2:  #include <iostream>
 3:
 4:  using namespace std;
 5:  void swap(int x, int y);
 6:
 7:  int main()
 8:  {
 9:     int x = 5, y = 10;
10:
11:     cout << "Dans main(). Avant échange, x : "
11a:             << x << ", y : " << y << endl;
12:     swap(x,y);
13:     cout << "Dans main(). Après échange, x : "
13a:             << x << ", y : " << y << endl;
14:     return 0;
15:  }
16:
17:  void swap (int x, int y)
18:  {
19:     int temp;
20:
21:     cout << "Dans swap. Avant échange, x : "
21a:             << x << ", y : " << y << endl;
22:
23:     temp = x;
24:     x = y;
25:     y = temp;
26:
27:     cout << "Dans swap. Après échange, x : "
27a:             << x << ", y : " << y << endl;
28:  }
```

Ce programme produit le résultat suivant :

```
Dans main(). Avant échange. x : 5, y :10
Dans swap. Avant échange, x : 5, y : 10
Dans swap. Après échange, x : 10, y : 5
Dans main(). Après échange. x : 5, y :10
```

Ce programme initialise deux variables dans la fonction principale, puis les passe à la fonction `swap()` qui les intervertit. De retour dans la fonction `main()`, leur contenu n'a pas été modifié !

L'initialisation et l'affichage des variables s'effectuent aux lignes 9 et 11. La fonction `swap()` est appelée à la ligne 12 et les variables lui sont passées en paramètre.

La fonction affiche à nouveau les variables à la ligne 21. Comme l'on pouvait s'y attendre, elles apparaissent dans le même ordre que dans la fonction principale. Puis elles sont interverties aux lignes 23 à 25, ce qui est confirmé par l'affichage de la ligne 27. Dans la fonction `swap()`, les deux valeurs sont donc bien échangées.

Le programme revient ensuite à la ligne 13. Dans la fonction `main()`, les valeurs n'ont pas été échangées.

Comme vous l'avez deviné, les paramètres sont passés *par valeur* à la fonction appelée, ce qui signifie qu'il s'agit de copies locales à la fonction. Ce sont donc ces copies qui ont été échangées aux lignes 23 à 25, pas les originales.

Aux Chapitres 8 et 10, vous verrez qu'il existe d'autres solutions que celle du passage par valeur, qui permettent de modifier les valeurs présentes dans `main()`.

Variables globales

Les variables définies en dehors de toute fonction ont une portée globale et sont donc disponibles à partir de n'importe quelle fonction du programme, y compris `main()`.

Une variable locale peut porter le même nom qu'une variable globale : la modification de son contenu ne modifiera pas la variable globale, mais la variable locale *masquera* la variable globale (voir le Listing 5.5). Lorsqu'une fonction possède une variable locale de même nom qu'une variable globale, ce nom fait référence à la variable locale lorsqu'elle est utilisée dans la fonction.

Listing 5.5 : Utilisation de variables locales et de variables globales

```
 1:  #include <iostream>
 2:  void maFonction();           // prototype
 3:
 4:  int x = 5, y = 7;            // variables globales
 5:  int main()
 6:  {
 7:     using namespace std;
 8:
 9:     cout << "x dans main : " << x << endl;
10:     cout << "y dans main : " << y << endl << endl;
11:     maFunction();
12:     cout << "De retour de maFonction ! << endl << endl;
13:     cout << "x dans main : " << x << endl;
14:     cout << "y dans main : " << y << endl;
15:     return 0;
```

```
16:  }
17:
18:  void maFonction()
19:  {
20:     using std::cout;
21:
22:     int y = 10;
23:
24:     cout << "x dans maFonction : " << x << endl;
25:     cout << "y dans maFonction : " << y << endl << endl;
26:  }
```

Ce programme produit le résultat suivant :

```
x dans main : 5
y dans main : 7

x dans maFonction : 5
y dans maFonction : 10

De retour de maFonction !

x dans main : 5
y dans main : 7
```

Ce programme simple met en évidence la confusion que peuvent causer variables locales et variables globales. La ligne 4 déclare et initialise les deux variables globales `x` et `y` avec les valeur 5 et 7.

Ces valeurs s'affichent alors que le programme est encore dans la fonction `main()`. Vous pouvez constater que la fonction principale ne définit pas les variables puisque `x` et `y` existent au niveau global.

La ligne 11 appelle la fonction `maFonction()`. Le programme saute directement à la ligne 18 et lit la première instruction de cette fonction. La variable locale `y` est initialisée à 10. Elle se substitue à la variable globale de même nom. À la ligne 25, lorsque la variable `y` est affichée, c'est donc le contenu de la variable locale qui apparaît et qui masque la variable globale de même nom.

Lorsque la fonction prend fin, le programme redonne la main à la fonction principale et tient de nouveau compte des variables globales `x` et `y`. Vous pouvez remarquer que la variable globale `y` n'a pas été affectée par la modification de la variable `y` locale à `maFonction()`.

Variables globales : quelques conseils

Bien que les variables globales soient autorisées par le langage, on ne les utilise quasiment jamais en C++. En C, elles sont dangereuses mais nécessaires lorsque l'on souhaite partager des données entre plusieurs fonctions sans les passer constamment en paramètre, notamment lorsqu'il s'agit simplement de les transmettre d'une fonction à l'autre.

Les variables globales sont dangereuses, car ce sont des données partagées et parce qu'une fonction peut modifier une variable globale sans que les autres fonctions ne le voient, ce qui peut produire des bogues très difficiles à détecter.

En C++, il est possible de remplacer les variables globales par des *variables membres statiques*. Pour en savoir plus, reportez-vous au Chapitre 15.

Instructions des fonctions

En théorie, les instructions dans le corps d'une fonction ne sont limitées ni en nombre ni en type. Bien que vous n'ayez pas le droit de définir une fonction à l'intérieur d'une fonction, vous pouvez *appeler* une fonction, ce qui est courant dans la fonction principale `main()`. Les fonctions peuvent s'appeler elles-mêmes, ce que nous verrons bientôt dans la section consacrée à la *récursivité*.

Bien que sa taille ne soit pas limitée, une fonction C++ bien conçue est de taille raisonnable. Certains programmeurs limitent la taille de chaque fonction à un écran, ce qui leur permet de visualiser toutes les instructions simultanément. Cette règle n'est pas stricte, mais il est vrai qu'une fonction courte est plus facile à comprendre et à mettre à jour.

Une fonction ne devrait être consacrée qu'à une seule tâche simple et facile à comprendre. Une fonction longue et complexe peut toujours se décomposer en fonctions courtes et simples.

Retour sur les paramètres des fonctions

Toute expression valide de C++ peut être un paramètre de fonction, y compris les constantes, les expressions mathématiques et logiques et les autres fonctions qui renvoient une valeur. Le point important est que le type de la valeur renvoyée par l'expression doit correspondre au type du paramètre attendu par la fonction.

Il est même possible d'utiliser la valeur renvoyée par une fonction comme paramètre d'une autre fonction. Toutefois, cela peut rendre le code correspondant difficile à lire et à déboguer.

Prenons les fonctions `monDouble()`, `triple()`, `carre()` et `cube()`, qui renvoient chacune une valeur. Vous pourriez écrire l'expression suivante :

```
resultat = (monDouble(triple(carre(cube(maValeur)))));
```

Vous pouvez aborder cette instruction de deux manières. D'une part, la fonction `monDouble()` prend la fonction `triple()` comme paramètre. À son tour, `triple()` prend la fonction `carre()`, qui prend la fonction `cube()` comme paramètre. La fonction `cube()` prend la variable `maValeur` comme paramètre.

Si l'on part dans l'autre sens, cette instruction prend la variable `maValeur` et la passe en paramètre à la fonction `cube()`, la valeur renvoyée est passée en paramètre à la fonction `carre()`, et ainsi de suite. La valeur obtenue à la fin est copiée dans `resultat`.

Vous remarquerez qu'il est assez difficile de déterminer les priorités d'exécution de cette expression.

Pour mieux comprendre ce traitement et rendre le code plus lisible, il est donc préférable de créer une variable intermédiaire recevant le résultat de chaque appel de fonction :

```
unsigned long maValeur  = 2;
unsigned long monCube   = cube(maValeur);       // monCube = 8
unsigned long monCarre  = carre(monCube);       // monCarre = 64
unsigned long monTriple = triple(monCarre);     // monTriple = 192
unsigned long resultat  = monDouble(monTriple); // Resultat = 384
```

Le programme est plus explicite !



Bien que C++ facilite l'écriture d'un code compact (voir l'exemple précédent) pour combiner les fonctions `cube()`, `carre()`, `triple()` et `monDouble()`, cette possibilité n'est pas une obligation. Il vaut mieux que le code soit simple à lire, donc à entretenir, que compact.

Retour sur les valeurs renvoyées

Les fonctions renvoient une valeur ou ne renvoient rien, auquel cas le type du résultat est `void`.

Pour renvoyer une valeur, il suffit d'entrer le mot-clé `return` suivi de la valeur à récupérer. Cette valeur peut être, elle-même, une expression qui produit une valeur :

```
return 5;
return (x > 5);
return (Fonction());
```

Ces trois instructions `return` sont valides, à condition que `Fonction()` renvoie une valeur. La valeur de la deuxième instruction `return (x > 5)` est égale à `true` ou à `false` selon que `x` est supérieur à 5 ou inférieur ou égal à 5. La valeur renvoyée n'est pas égale à `x`, mais au résultat du test.

Lorsque le mot-clé `return` est détecté, l'expression située immédiatement après est renvoyée comme valeur de la fonction. Le code qui a appelé la fonction reprend alors la main et les instructions qui sont éventuellement placée après le mot-clé `return` ne sont pas prises en compte.

Une fonction peut contenir une ou plusieurs instructions `return`, comme le montre le Listing 5.6.

Listing 5.6 : Utilisation de plusieurs instructions *return*

```
 1:  // Listing 5.6 - Plusieurs instructions return
 2:  // Démonstration
 3:  #include <iostream>
 4:
 5:  int Doubleur(int montantADoubler);
 6:
 7:  int main()
 8:  {
 9:     using std::cout;
10:
11:     int resultat = 0;
12:     int entree;
13:
14:     cout <<
14a:   "Entrez un nombre entre 0 et 10000 à multiplier par 2 : ";
15:     std::cin >> entree;
16:
17:     cout << "\nAvant d'appeler la fonction Doubleur()... ";
18:     cout << "\nValeur entrée : " << entree <<
18a:            ", Valeur doublée : " << resultat << "\n";
19:
20:     resultat = Doubleur(entree);
21:
22:     cout << "\nDe retour de la fonction Doubleur()...\n";
23:     cout << "\nValeur entrée : " << entree <<
23a:            ", Valeur doublée : " << resultat << "\n";
24:
25:     return 0;
26:  }
```

```
27:
28:  int Doubleur(int original)
29:  {
30:     if (original <= 10000)
31:        return original * 2;
32:     else
33:        return -1;
34:     std::cout << "Vous n'atteindrez jamais cette ligne !\n";
35:  }
```

Ce programme produit le résultat suivant :

```
Entrez un nombre entre 0 et 10000 à multiplier par 2 : 9000

Avant d'appeler la fonction Doubleur()...
Valeur entrée : 9000, Valeur doublée : 0

De retour de la fonction Doubleur() ...

Valeur entrée : 9000, Valeur doublée : 18000

Entrez un nombre entre 0 et 10000 à multiplier par 2 : 11000

Avant d'appeler la fonction Doubleur()...
Valeur entrée : 11000, Valeur doublée : 0

De retour de la fonction Doubleur()...
Valeur entrée : 11000, Valeur doublée : -1
```

Vous êtes invité à entrer un nombre (lignes 14 et 15) dans une variable locale que l'on affiche ensuite, accompagnée de la valeur courante de `resultat` (lignes 17 et 18). La ligne 20 appelle la fonction `Doubleur()` en lui passant en paramètre la valeur saisie. Le résultat renvoyé par la fonction est copié dans la variable locale `resultat` et les valeurs sont de nouveau affichées au niveau de la ligne 23.

Dans la fonction `Doubleur()` (ligne 30), le programme détermine si le paramètre est inférieur ou égal à 10 000. Si c'est le cas, le nombre d'origine est renvoyé après avoir été doublé. Dans le cas contraire, la valeur renvoyée `-1` indique à une erreur.

L'exécution de la fonction n'atteint jamais la ligne 34 car elle s'arrête à la ligne 31 ou à la ligne 33, quelle que soit la valeur saisie. Un bon compilateur détectera l'erreur et émettra un message d'avertissement… et un bon développeur supprimera cette ligne inutile !

Quelle est la différence entre `int main()` et `void main();` ? Quelle forme faut-il utiliser ? J'ai utilisé les deux avec succès, pourquoi faut-il écrire : `int main(){ return 0;}` ?

Les deux fonctionnent sur la plupart des compilateurs, mais seul `int main()` est compatible ANSI, il est donc assuré de continuer à fonctionner.

La différence est la suivante : `int main()` renvoie une valeur au système d'exploitation. Lorsque votre programme se termine, cette valeur peut être testée dans un script, par exemple.

Dans ce livre, nous n'utilisons pas la valeur renvoyée, mais le standard ANSI exige que `main()` en renvoie une.

Paramètres par défaut

À chaque paramètre déclaré dans un prototype ou dans une définition correspond une valeur que le programme doit passer à la fonction. Bien entendu, le type de valeur doit être identique à celui qui a été déclaré. Exemple :

```
long maFonction(int);
```

Ici, la fonction attend une variable entière. Si la définition ne correspond pas ou si le type est différent, le compilateur produit une erreur et ne crée pas de fichier objet.

Les prototypes qui déclarent une valeur par défaut pour le paramètre font exception. Une valeur par défaut est une valeur utilisée lorsque aucune autre n'est fournie. La déclaration figurant plus haut pourrait être réécrite ainsi :

```
long maFonction (int x = 50);
```

Ce prototype signifie que la fonction renvoie un entier long et quel attend un paramètre entier. Si aucun paramètre ne lui est transmis lors de l'appel, elle utilisera 50 comme valeur par défaut. Rappelons que les noms de paramètres n'étant pas obligatoires dans les prototypes de fonctions, on pourrait également écrire :

```
long maFonction (int = 50);
```

La définition de la fonction n'est pas modifiée par la présence d'un paramètre par défaut et son en-tête se présenterait donc de la façon suivante :

```
long maFonction (int x)
```

Si la fonction appelante ne fournit pas de valeur à l'appel de `maFonction()`, le nombre 50 sera copié dans le paramètre `x`. Il n'est pas nécessaire de donner le même nom au paramètre

par défaut dans le prototype et dans l'en-tête de fonction, car la valeur par défaut est affectée en fonction de sa position, pas de son nom.

Vous pouvez fournir des valeurs par défaut selon vos besoins. La seule règle que vous devez respecter est que si un paramètre n'a pas de valeur par défaut, aucun des paramètres qui le précèdent ne peut en avoir (en d'autres termes, les paramètres avec des valeurs par défaut doivent apparaître à la fin de la liste).

Si le prototype de la fonction est de la forme :

```
long maFonction (int Param1, int Param2, int Param3);
```

`Param2` ne peut avoir une valeur par défaut que si `Param3` en a également une. De même, `Param1` ne peut avoir une valeur par défaut que si vous en avez affecté une aux deux autres. Le Listing 5.7 illustre cette règle.

Listing 5.7 : Valeurs par défaut des paramètres

```
 1:  // Listing 5.7 - Utilisation des
 2:  // valeurs de paramètres par défaut
 3:  #include <iostream>
 4:
 5:  int VolumeCube
5a:     (int longueur, int largeur = 25, int hauteur = 1);
 6:
 7:  int main()
 8:  {
 9:     int longueur = 100;
10:     int largeur = 50;
11:     int hauteur = 2;
12:     int volume;
13:
14:     volume = VolumeCube(longueur, largeur, hauteur);
15:     std::cout << "Le premier volume est égal à : "
15a:              << volume << "\n";
16:
17:     volume = VolumeCube(longueur, largeur);
18:     std::cout << "Le deuxième volume est égal a : "
18a:              << volume << "\n";
19:
20:     volume = VolumeCube(longueur);
21:     std::cout << "Le troisième volume est égal à : "
21a:              << volume << "\n";
22:     return 0;
23:  }
24:
25:  int VolumeCube(int longueur, int largeur, int hauteur)
```

```
26:  {
27:
28:     return (longueur * largeur * hauteur);
29:  }
```

Ce programme produit le résultat suivant :

```
Le première volume est égal à : 10000
Le deuxième volume est égal à : 5000
Le troisième volume est égalà : 2500
```

À la ligne 5, le prototype `VolumeCube()` indique que la fonction de même nom attend trois paramètres entiers, dont les deux derniers ont une valeur par défaut.

La fonction calcule le volume du cube dont on lui a passé les dimensions. Si on ne lui fournit pas de largeur explicite, elle utilise une largeur de 25 et une hauteur de 1. Il est impossible de passer la hauteur sans passer aussi une largeur.

Aux lignes 9, 10 et 11, on initialise les variables `longueur`, `largeur` et `hauteur`, qui sont ensuite passées à la fonction `VolumeCube()` en ligne 14. Le résultat de ce calcul est affiché à la ligne 15.

À la ligne 17, le programme reprend la main et rappelle la fonction. Cette fois-ci, aucune valeur n'est fournie pour la hauteur et c'est donc sa valeur par défaut qui est prise en compte. Le calcul s'effectue et le résultat est de nouveau affiché.

L'exécution se poursuit à la ligne 20, cette fois-ci en ne fournissant ni la largeur ni la hauteur. Les valeurs par défaut correspondantes sont donc utilisées et l'on affiche le résultat obtenu.

Faire	*Ne pas faire*
● Considérer les paramètres comme des variables locales dont la portée se limite à la fonction appelée. ● Se souvenir que la modification d'une variable globale dans une fonction est répercutée dans toutes les fonctions.	● Créer une valeur par défaut pour un paramètre s'il n'en existe pas pour le paramètre suivant. ● Oublier qu'un paramètre passé par valeur ne modifie pas la variable correspondante dans la fonction appelante.

Surcharge de fonctions

C++ permet de créer plusieurs fonctions portant le même nom. Cette fonctionnalité est appelée *surcharge de fonction*. Une fonction surchargée doit être distincte au niveau de la

liste de ses paramètres, qui doit différer par le type et/ou de le nombre des paramètres. Exemple :

```
int Fonction (int, int);
int Fonction (long, long);
int Fonction (long);
```

La fonction `Fonction()` est surchargée avec trois listes de paramètres. Dans les deux premières versions, les paramètres changent de type. Dans la dernière, la liste des paramètres ne contient pas le même nombre de paramètres.

Le type des valeurs renvoyées par les fonctions surchargées peut être identique ou non.



Lorsque deux fonctions renvoient des valeurs de types différents alors que leur nom et leur liste de paramètres sont identiques, on obtient une erreur de compilation. Pour modifier le type du résultat, vous devez également modifier la signature (nom et/ou liste de paramètres).

La surcharge de fonctions est aussi appelée *polymorphisme de fonction*. *Poly* signifie plusieurs et *morphe* signifie forme : une fonction polymorphe pourra donc prendre plusieurs formes.

Le polymorphisme de fonction représente la capacité de "surcharger" une fonction avec plusieurs significations. Comme nous l'avons vu, il est possible de modifier le nombre et le type des paramètres en surchargeant une fonction. Entre deux fonctions surchargées, le compilateur déterminera celle qui doit être appelée en fonction du type et/ou du nombre des paramètres utilisés. Ainsi, grâce à la surcharge, vous pouvez créer la fonction `Moyenne()` capable de calculer tour à tour la moyenne de valeurs entières, la moyenne de valeurs fractionnaires, etc., ce qui vous évite de devoir définir les fonctions `MoyenInt()`, `MoyenFloat()`, etc.

Supposons que vous définissiez une fonction qui multiplie par 2 la valeur saisie. Il sera souhaitable qu'elle puisse calculer des valeurs de tout type (`int`, `float` ou `double`). Si vous n'utilisez pas la surcharge de fonction, vous devrez créer quatre fonctions distinctes :

```
int DoubleInt(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

Grâce à la surcharge, il suffira de déclarer la même fonction, de la façon suivante :

```
int Double(int);
long Double(long);
```

```
float Double(float);
double Double(double);
```

ce qui est bien plus facile à lire et à utiliser. Vous n'avez plus besoin de savoir quelle fonction appeler : il suffira de passer une valeur en paramètre et le programme appellera automatiquement la bonne fonction, comme le montre le Listing 5.8.

Listing 5.8 : Exemple de fonction polymorphe

```
 1:  // Listing 5.8 - Exemple de
 2:  // fonction polymorphe
 3:  #include <iostream>
 4:
 5:  int Double(int);
 6:  long Double(long);
 7:  float Double(float);
 8:  double Double(double);
 9:
10:  using namespace std;
11:
12:  int main()
13:  {
14:     int      monInt = 6500;
15:     long     monLong = 65000;
16:     float    monFloat = 6.5F;
17:     double   monDouble = 6.5e20;
18:
19:     int      doubleInt;
20:     long     doubleLong;
21:     float    doubleFloat;
22:     double   doubleDouble;
23:
24:     cout << "monInt : " << monInt << "\n";
25:     cout << "monLong : " << monLong << "\n";
26:     cout << "monFloat : " << monFloat << "\n";
27:     cout << "monDouble : " << monDouble << "\n";
28:
29:     doubleInt = Double(monInt);
30:     doubleLong = Double(monLong);
31:     doubleFloat = Double(monFloat);
32:     doubleDouble = Double(monDouble);
33:
34:     cout << "doubleInt : " << doubleInt << "\n";
35:     cout << "doubleLong : " << doubleLong << "\n";
36:     cout << "doubleFloat : " << doubleFloat << "\n";
37:     cout << "doubleDouble : " << doubleDouble << "\n";
```

```
38:
39:     return 0;
40:  }
41:
42:  int Double(int original)
43:  {
44:     cout << "Dans Double(int)\n";
45:     return 2 * original;
46:  }
47:
48:  long Double(long original)
49:  {
50:     cout << "Dans Double(long)\n";
51:     return 2 * original;
52:  }
53:
54:  float Double(float original)
55:  {
56:     cout << "Dans Double(float)\n";
57:     return 2 * original;
58:  }
59:
60:  double Double(double original)
61:  {
62:     cout << "Dans Double(double)\n";
63:     return 2 * original;
64:  }
```

Ce programme produit le résultat suivant :

```
monInt : 6500
monLong : 65000
monFloat : 6.5
monDouble : 6.5e+20
Dans Double(int)
Dans Double(long)
Dans Double(float)
Dans Double(double)
DoubleInt : 13000
DoubleLong : 130000
DoubleFloat : 13
DoubleDouble : 1.3e+21
```

La fonction `Double()` est surchargée avec `int`, `long`, `float` et `double`. Les prototypes figurent de la ligne 5 à la ligne 8, les définitions de la ligne 42 à la ligne 64.

Notez que, dans cet exemple, l'instruction `using namespace std;` a été ajoutée à la ligne 10, hors de toute fonction spécifique. L'instruction est donc globale au fichier et l'espace de nom est utilisé dans toutes les fonctions déclarées.

Le corps du programme contient huit variables locales. De la ligne 14 à la ligne 17, quatre valeurs sont initialisées alors que les quatre autres reçoivent le résultat de la fonction `Double()` (lignes 29 à 32). Lorsque la fonction est appelée, c'est le paramètre qui est passé qui permet au programme de déterminer la bonne version.

Pour cela, le compilateur examine les paramètre, puis appelle l'une des quatre fonctions `Double()` surchargées. Le résultat affiché montre qu'il ne s'est pas trompé !

Pour en savoir plus sur les fonctions

Les fonctions étant un aspect crucial de la programmation, certains points particuliers méritent d'être connus pour résoudre des problèmes inhabituels. Utilisées correctement, les fonctions *en ligne* permettent d'extraire la dernière goutte de performances d'un programme. La récursivité, quant à elle, fait partie de ces merveilles ésotériques de la programmation et permet de résoudre aisément des problèmes qui pourraient pas être traités aussi simplement sans elle.

Fonctions en ligne

Lorsque l'on définit une fonction, le compilateur produit généralement un seul ensemble d'instructions en mémoire. Lorsque cette fonction est appelée, le programme saute directement au début de ces instructions et, quand la fonction se termine, il revient à la ligne qui suit immédiatement l'appel. Si vous invoquez dix fois la fonction, le programme exécutera autant de fois les mêmes instructions, mais il n'existera en fait qu'une seule copie de la fonction dans le code (et non 10).

Ces sauts et ces retours ont un petit impact sur les performances. Certaines fonctions sont très courtes puisqu'elles n'ont qu'une ou deux lignes de code : il serait alors plus efficace pour le programme d'éviter de se brancher à un autre emplacement mémoire uniquement pour exécuter une ou deux instructions. En programmation, efficacité rime avec rapidité : un programme s'exécute plus vite s'il y a moins d'appels de fonctions.

C++ propose une solution. Lorsqu'une fonction est déclarée avec le mot-clé `inline`, le compilateur ne crée pas une véritable fonction et se contente de remplacer l'appel de cette fonction par son code dans la fonction appelante. Il n'y a donc plus de saut en mémoire lors de l'appel : c'est exactement comme si vous aviez directement écrit les instructions de la fonction dans la fonction appelante.

Cela dit, les fonctions en ligne peuvent coûter cher. Si la fonction est appelée dix fois, le code `inline` sera copié autant de fois dans les fonctions appelantes. Cet accroissement de la taille du fichier exécutable peut complètement occulter le gain minime de rapidité obtenu et, en fait, le programme peut parfois s'en trouver ralenti !

De nos jours, les compilateurs peuvent quasiment toujours faire un meilleur travail que vous pour prendre cette décision lorsqu'elle s'impose et il est généralement préférable de ne pas déclarer une fonction `inline`, sauf si elle ne comporte qu'une ou deux instructions. Dans le doute, ne le faites pas.



L'optimisation des performances est une opération difficile et la plupart des programmeurs ont du mal à identifier les problèmes de leurs programmes sans aide, ce qui implique des programmes spécialisés, comme les débogueurs et les profileurs.
De même, il vaut toujours mieux écrire du code clair et compréhensible plutôt que supposer qu'il sera lent ou rapide, quitte à écrire un code difficile à comprendre. En effet, il est toujours plus facile d'accélérer un code clair.

Listing 5.9 : Exemple de fonction en ligne

```
// Listing 5.9 - Exemple de fonction en ligne
#include <iostream>

inline int Double(int);

int main()
{
   int cible;
   using std::cout;
   using std::cin;
   using std::endl;

   cout << "Entrez un nombre : ";
   cin >> cible;
   cout << "\n";

   cible = Double(cible);
   cout << "Valeur cible : " << cible << endl;

   cible = Double(target);
   cout << "Valeur cible : " << cible << endl;

   cible = Double(target);
   cout << "Valeur cible : " << cible << endl;
```

```
25:     return 0;
26:  }
27:
28:  int Double(int cible)
29:  {
30:     return 2*cible;
31:  }
```

Ce programme produit le résultat suivant :

```
Entrez un nombre : 20

Valeur cible : 40
Valeur cible : 80
Valeur cible : 160
```

La ligne 4 déclare la fonction `Double() inline` avec un paramètre et type de résultat qui sont tous deux des `int`. Cette déclaration ressemble à n'importe quel autre prototype, sauf que le mot-clé `inline` précède le type du résultat.

La compilation revient à produire le code suivant :

```
cible = 2 * cible;
```

à chaque fois que le programme va rencontrer :

```
cible = Double(cible);
```

Au moment de l'exécution, les instructions de la fonction ont été intégrées dans le code, compilées dans le fichier `.obj`. Cela évite un branchement et un retour au cours de l'exécution du programme, au prix d'un fichier exécutable plus volumineux.



`inline` est un mot-clé indiquant simplement au compilateur que vous voulez que la fonction soit en ligne. Le compilateur peut ne pas en tenir compte et réaliser un appel normal.

Récursivité

Une fonction peut s'appeler elle-même. Cette fonctionnalité s'appelle la *récursivité*. Elle peut être directe ou indirecte. Elle est *directe* lorsque la fonction s'appelle elle-même et *indirecte* lorsqu'une fonction appelle une autre fonction qui appelle à son tour la première.

Certains problèmes se résolvent plus simplement à l'aide de la récursivité : il s'agit généralement de ceux où l'on traite des données pour obtenir un résultat et que l'on applique le même traitement à ce résultat. Qu'elle soit directe ou indirecte, la récursivité peut se terminer et produire un résultat ou ne jamais finir et provoquer une erreur d'exécution.

Lorsqu'une fonction s'appelle elle-même, il faut bien comprendre que c'est une nouvelle copie de la fonction qui s'exécute. Les variables locales de cette copie sont totalement indépendantes de celles de la première et elles ne peuvent pas les modifier directement, exactement comme les variables locales de `main()` ne peuvent pas affecter les variables locales des fonctions qu'elle appelle, comme on l'a montré dans le Listing 5.3.

Nous allons utiliser la récursivité pour traiter la suite de Fibonacci :

```
1,1,2,3,5,8,13,21,34...
```

Après le deuxième, chaque nombre est la somme des deux précédents. Un problème de Fibonacci consiste, par exemple, à trouver le 12e nombre de la suite.

Pour résoudre ce problème, il faut soigneusement étudier la suite. Les deux premières valeurs sont égales à 1. Chacun des nombres qui suit est la somme des deux précédents. Le septième nombre, par exemple, correspond à la somme du sixième et du cinquième. Plus généralement, le nième nombre est la somme de `n-2` et de `n-1` pour `n > 2`.

Toute fonction récursive doit inclure une condition d'arrêt. En son absence, le programme ne se terminerait jamais. Dans notre exemple, la condition d'arrêt correspond à `n < 3` (c'est-à-dire que lorsque n est inférieur à 3, nous pouvons arrêter le traitement).

L'algorithme du programme se présente ainsi :

1. Entrer la position à trouver dans la suite.
2. Appeler la fonction `fib()`, en transmettant la valeur saisie.
3. La fonction `fib()` teste son paramètre (`n`). Si `n < 3`, elle renvoie `1` ; dans le cas contraire, elle s'appelle elle-même deux fois, en passant la valeur `n-2` puis la valeur `n-1`. Enfin, elle renvoie la somme des résultats des deux appels à la fonction principale.

Si vous appelez `fib(1)` ou `fib(2)`, la fonction renvoie 1 et s'arrête. Si vous appelez `fib(3)`, elle retournera la somme des valeurs calculées lors de l'appel à `fib(2)` et de l'appel à `fib(1)`, c'est-à-dire 2 (1+1).

De même, l'appel `fib(4)` renvoie la somme de l'appel de `fib(3)` et de `fib(2)`. Nous venons de voir que `fib(3)` renvoyait 2 (en appellant `fib(2)` et `fib(1)`) et que `fib(2)` renvoyait 1. Par conséquent, `fib(4)` additionne ces deux nombres et renvoie 3, qui est le quatrième nombre de la suite.

Avec une étape de plus, `fib(5)` renvoie la somme de `fib(4)` et `fib(3)`, soit 3 + 2, c'est-à-dire 5.

Cette méthode n'est pas la plus efficace pour résoudre le problème (l'appel `fib(20)` appellera 13 529 fois la fonction `fib()` avant de renvoyer le résultat !), mais elle fonctionne. Faites attention : si vous utilisez un paramètre trop grand, vous tomberez à cours de mémoire car, à chaque appel, la mémoire est sollicitée et n'est libérée qu'après la fin du

premier appel. Elle risque donc d'être rapidement saturée. Cette fonction est implémentée dans le Listing 5.10.



Dans le programme ci-après, évitez de choisir une valeur supérieure à 15, car la mémoire de votre PC risque ne pas être suffisante !

Listing 5.10 : La récursivité appliquée à la suite de Fibonacci

```
// Exemple de récursivité
#include <iostream>
int fib (int n);

int main()
{

   int n, reponse;
   std::cout << "Entrez le nombre a trouver : ";
   std::cin >> n;

   std::cout << "\n\n";

   reponse = fib(n);

   std::cout << reponse << " est le " << n;
      std::cout << "e nombre dans la suite\n";
   return 0;
}

int fib (int n)
{
   std::cout << "Traitement fib(" << n << ")... ";

   if (n < 3 )
   {
      std::cout << "Renvoie 1!\n";
      return (1);
   }
   else
   {
      std::cout << "Appel de fib(" << n-2 << ") ";
         std::cout << "et fib(" << n-1 << ").\n";
      return( fib(n-2) +  fib(n-1));
   }
}
```

Ce programme produit le résultat suivant :

```
Entrez le rang a trouver : 6

Traitement fib(6)... Appel de fib(4) et fib(5).
Traitement fib(4)... Appel de fib(2) et fib(3).
Traitement fib(2)... Renvoie 1 !
Traitement fib(3)... Appel de fib(1) et fib(2).
Traitement fib(1)... Renvoie 1 !
Traitement fib(2)... Renvoie 1 !
Traitement fib(5)... Appel de fib(3) et fib(4).
Traitement fib(3)... Appel de fib(1) et fib(2).
Traitement fib(1)... Renvoie 1 !
Traitement fib(2)... Renvoie 1 !
Traitement fib(4)... Appel de fib(2) et fib(3).
Traitement fib(2)... Renvoie 1 !
Traitement fib(3)... Appel de fib(1) et fib(2).
Traitement fib(1)... Renvoie 1 !
Traitement fib(2)... Renvoie 1 !
8 est le 6e nombre dans la suite
```



Certains compilateurs ont des difficultés avec l'usage des opérateurs dans une instruction `cout`. Si vous recevez un avertissement à la ligne 32, mettez des parenthèses autour de la soustraction pour que les lignes 32 et 33 deviennent :

```
std::cout << "Appel de fib(" << (n-2) << ") ";
std::cout << "et fib(" << (n-1) << ").\n";
```

À la ligne 9, l'utilisateur entre un nombre qui est copié dans `n`. Cette valeur est transmise à `fib()`. Le programme se branche sur la fonction `fib()`, comme vous pouvez le vérifier à l'écran (ligne 23).

À la ligne 25, le programme vérifie si l'argument `n` est inférieur à 3. Si tel est le cas, la fonction `fib()` renvoie la valeur 1. Dans le cas contraire, elle renvoie la somme des valeurs après avoir traité `n-2` et `n-1`.

Ces valeurs ne peuvent pas être renvoyées avant la fin de l'appel à `fib()`. Vous pouvez donc imaginez que le programme appelle `fib` de façon répétitive jusqu'à ce qu'un de ces appels renvoie une valeur. Les seuls qui renvoient une valeur sont les appels à `fib(2)` et `fib(1)`. Ces valeurs sont ensuite passées aux appelants en attente qui, à leur tour, ajoutent la valeur renvoyée à la leur propre, puis la retournent. Les Figures 5.4 et 5.5 illustrent ces appels récursifs à `fib()`.

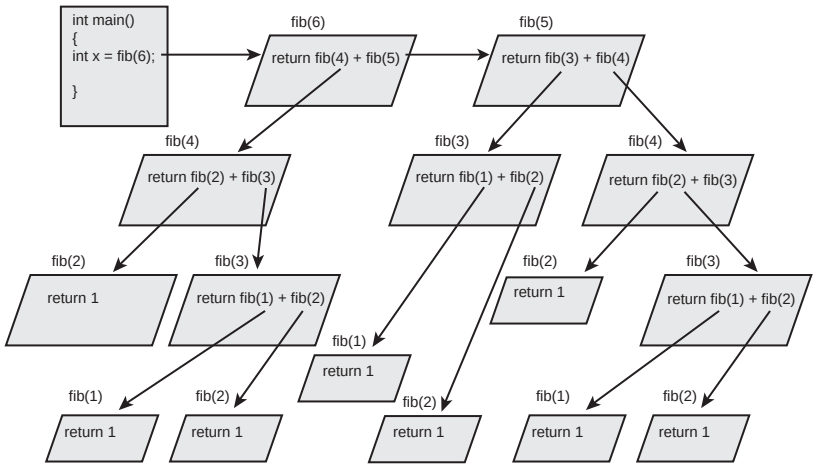


Figure 5.4
Utilisation de la récursivité.

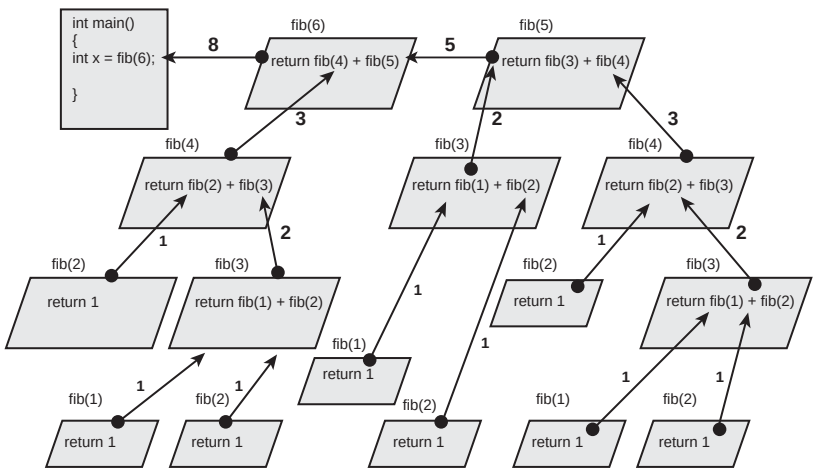


Figure 5.5
Retour après les appels récursifs.

Dans cet exemple, `n` est égal à 6 et la fonction `fib(6)` est appelée depuis la fonction principale. À la ligne 25, le programme vérifie si `n < 3`. Le test échoue et la fonction `fib(6)` renvoie à la ligne 34 la somme des valeurs renvoyées par `fib (4)` et `fib(5)`. Voici la ligne 34 :

```
return( fib(n-2) +  fib(n-1));
```

Un appel est fait à `fib(4)` depuis cette instruction de retour (comme `n == 6`, `fib(n-2)` est la même chose que `fib(4)`) et un autre appel est fait à `fib(5)` (`fib(n-1)`), puis la fonction dans laquelle vous vous trouvez (`fib(6)`) *attend* que ces appels renvoient une valeur. À ce moment, la fonction peut renvoyer le résultat de l'addition de ces deux valeurs.

Puisque `fib(5)` passe un argument qui n'est pas inférieur à 3, `fib()` sera de nouveau appelée, cette fois avec 4 et 3. `fib(4)` à son tour, appellera `fib(3)` et `fib(2)`.

Grâce à l'affichage des valeurs, vous pouvez suivre le déroulement du programme. À votre tour de tester le code. Lorsque le fichier exécutable est prêt, entrez 1, puis 2, puis 3 et allez jusqu'à 6 en étudiant soigneusement ce qui s'affiche.

Le moment est tout à fait propice au test de votre débogueur. Placez un point d'arrêt à la ligne 21, puis faites un *pas à pas détaillé* (ou *step into*) dans chaque appel à `fib`, en gardant la trace de la valeur de `n` à chaque appel récursif à `fib`.

La récursivité n'est pas très utilisée en programmation C++. Toutefois, elle s'avère très efficace pour résoudre certains problèmes.



La récursivité est un domaine difficile à maîtriser. Nous l'avons présentée parce qu'il fallait que vous connaissiez ce concept. Toutefois, ne vous inquiétez pas si certains détails vous ont échappé.

Principe des fonctions

Lors de l'appel d'une fonction, le programme se branche directement dans le code correspondant, les paramètres sont passés et le corps de la fonction s'exécute. Une fois terminée, la fonction renvoie une valeur (sauf si vous avez indiqué le mot-clé `void`) et rend la main à la fonction appelante.

Comment cela est-il possible ? Comment le code sait-il où se brancher ? Où sont stockées les variables qui sont transmises à la fonction ? Qu'arrive-t-il aux variables qui sont définies dans la fonction ? Comment la valeur renvoyée est-elle retransmise au programme ? Comment le programme sait-il où il doit reprendre ?

La plupart des manuels d'initiation à la programmation n'abordent pas ces questions et le développement garde ses secrets. Pour comprendre comment cela se passe, immergeons-nous dans la mémoire de l'ordinateur.

Les niveaux d'abstraction de la programmation

Au début de sa carrière, un programmeur manque de méthodologie et gère difficilement les niveaux d'abstraction intellectuelle du développement. Bien entendu, les ordinateurs sont des machines électroniques totalement dénuées d'intelligence. Ils ne "connaissent" rien des fenêtres et des menus, ils ne savent rien des programmes ou des instructions et ne connaissent même pas les zéros et les uns, puisqu'ils fonctionnent par impulsions électriques au niveau des circuits imprimés... ce qui est aussi intrinsèquement une abstraction.

Certains programmeurs refusent de connaître en détail la structure de la mémoire vive, prétextant qu'il n'est pas utile d'être expert en mécanique pour conduire une voiture, par exemple.

Vous devez comprendre l'organisation de la mémoire vive. Il est indispensable de savoir comment les variables sont créées et gérées et dans quelle partie les paramètres sont transférés.

Le partitionnement de la RAM

Lorsque vous créez un programme, le système d'exploitation (DOS, Linux/UNIX ou Windows, par exemple) configure les différentes zones de la mémoire en fonction des informations fournies par le compilateur. Comme vous êtes un développeur C++, vous allez entendre parler d'espace de noms global, de mémoire libre, de registres, d'espace de code et de pile.

Les variables globales sont dans l'espace de noms global. Vous découvrirez les notions d'espace de noms global et d'espace libre dans les chapitres suivants. Pour le moment, nous allons nous intéresser aux registres, à l'espace de code et à la pile.

Les registres sont une zone de mémoire spéciale, intégrée à l'unité centrale (également appelée UC ou *CPU*). Ils gèrent le fonctionnement interne de l'ordinateur. Leur mode de traitement est complexe et dépasse de loin le cadre de cet ouvrage mais les registres qui nous intéressent sont ceux qui pointent en permanence sur la ligne suivante de code et qui forment ce que l'on appelle le pointeur d'instruction. Un pointeur d'instruction garde la trace de la prochaine ligne de code à exécuter.

Les instructions se trouvent dans l'espace de code, qui est la partie de la mémoire qui stocke le code binaire produit par le compilateur. Chaque ligne de code source est traduite en une suite d'instructions, chacune d'elles occupant une adresse particulière en mémoire. Le pointeur d'instruction contient l'adresse de la prochaine instruction à traiter. Tout cela est représenté dans la Figure 5.6.

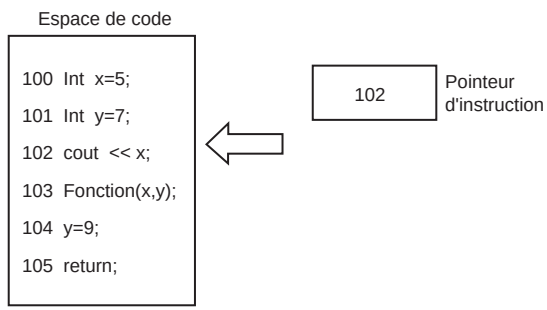
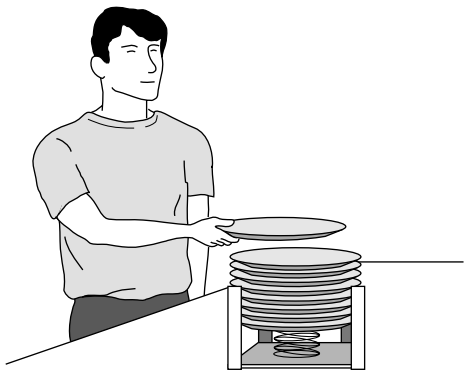


Figure 5.6
Le pointeur d'instruction.

La pile est une zone spéciale de mémoire allouée au programme, dans laquelle celui-ci organise les données utilisées par les fonctions. On l'appelle pile car c'est une structure de donnée LIFO (*Last In, First Out*, soit "dernier entré, premier sorti"), que l'on pourrait comparer à une pile d'assiettes (voir Figure 5.7).

Figure 5.7
Une pile.



Comme pour une pile d'assiettes ou un tas de pièces, le dernier élément posé est le premier retiré. Cela diffère de la plupart des files d'attente dans lesquelles le premier arrivé est le premier sorti.

La taille de la pile varie avec le nombre d'éléments qui entrent, puis sortent. Lorsque l'on *empile* des assiettes, la pile grossit ; lorsqu'on *dépile* des assiettes, elle se réduit. Il est impossible de retirer une assiette située à la base d'une pile sans retirer d'abord toutes les assiettes posées dessus.

Une pile d'assiettes est une analogie classique, mais il serait plus juste de comparer une pile à un ensemble de cases superposées de haut en bas. Le sommet de la pile est la case sur laquelle pointe le pointeur de pile (qui est également un registre).

Chaque case est associée à une adresse séquentielle, l'une d'elles étant stockée dans le registre du pointeur de pile. Toutes les adresses situées en dessous de cette adresse "magique" appartiennent à la pile. Tout ce qui est au-dessus du sommet se trouve en dehors de la pile et est considéré comme non valide (voir Figure 5.8).

Lorsque des données sont posées sur la pile, elles sont placées dans une case située au-dessus du pointeur, puis ce dernier se déplace vers le haut pour pointer sur ces nouvelles données. Lorsque des données sont ôtées de la pile, le pointeur se déplace simplement sur l'adresse précédente (voir Figure 5.9).

Les données *au-dessus* du pointeur de pile (hors de la pile) peuvent ou non être changées à tout moment. Ces valeurs sont désignées sous le nom de *garbage* (informations incohérentes) car leurs valeurs ne sont plus fiables.

Figure 5.8
Le pointeur de pile.

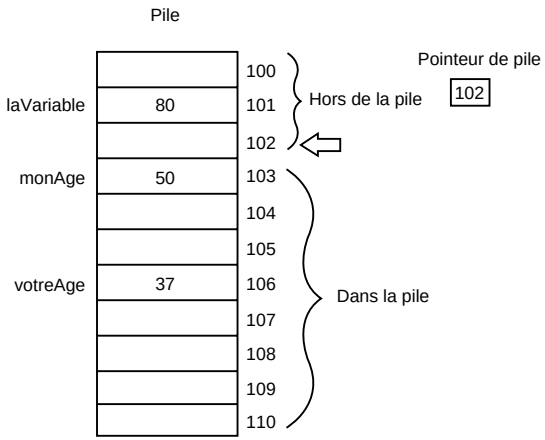
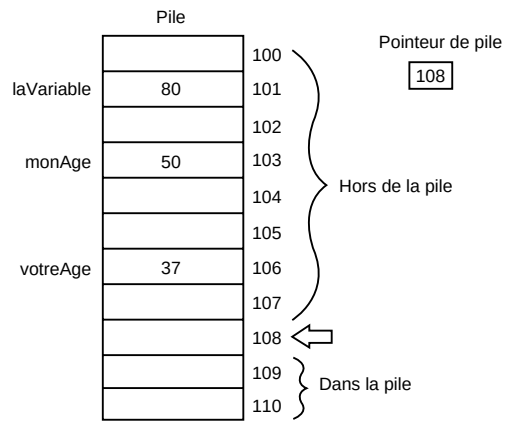


Figure 5.9
Déplacement du pointeur de pile.



La pile et les fonctions

Ce qui suit est une approximation de ce qui se passe lorsque votre programme se branche à une fonction. Les détails peuvent différer en fonction du système d'exploitation et du compilateur.

1. L'adresse dans le pointeur d'instruction devient celle qui suit l'appel de la fonction. Elle est placée sur la pile, ce sera l'adresse de retour après la fin de la fonction.
2. La pile est préparée pour recevoir le type de la valeur renvoyée : si ce type est un `int` et qu'un `int` occupe deux octets sur l'ordinateur, deux octets vides sont ajoutés au sommet de la pile, mais ne sont pas inititalisés (cela signifie que ce qu'ils contiennent y restera jusqu'à ce que la variable locale soit initialisée).
3. L'adresse de la fonction appelée, conservée dans une zone spéciale de la mémoire, est chargée dans le pointeur d'instruction. La prochaine instruction exécutée sera donc la première instruction de la fonction appelée.

4. Le sommet courant de la pile est marqué à l'aide d'un pointeur spécial appelé *stack frame*. Tout ce qui sera ajouté à la pile jusqu'au retour de la fonction aura désormais une portée *locale*, c'est-à-dire limitée à la fonction.
5. Tous les paramètres passés à la fonction sont placés sur la pile.
6. L'instruction présente dans le pointeur d'instruction s'exécute. Il s'agit de la première instruction de la fonction appelée.
7. Les variables locales sont empilées sur la pile au fur et à mesure qu'elles sont définies.

Quand la fonction est prête à se terminer, la valeur de son résultat est placée dans la zone de la pile réservée à l'étape 2. Puis la pile est dépilée jusqu'au *stack frame*, ce qui revient à faire disparaître toutes les variables locales et les paramètres passés à la fonction.

La valeur renvoyée est dépilée de la pile et devient la valeur de l'appel de la fonction. L'adresse de l'étape 1 est récupérée et placée dans le pointeur d'instruction. Le programme reprend alors son déroulement après l'instruction d'appel de la fonction.

Bien entendu, les détails du processus varient d'un compilateur à un autre et en fonction des systèmes d'exploitation ou des processeurs. Néanmoins, l'organisation de la pile est sensiblement la même sur tous les systèmes d'exploitation.

Au cours des prochains chapitres, nous aborderons d'autres emplacements mémoire qui servent à stocker des données qui doivent persister en dehors de la vie des fonctions.

Questions-réponses

Q Pourquoi ne pas utiliser que des variables globales ?

R Il y a quelques années, c'était la règle. Cependant, à mesure que les programmes sont devenus plus complexes, il est devenu difficile de trouver les bogues dus à des variables pouvant être modifiées par n'importe quelle fonction. En effet, les variables globales peuvent être modifiées de n'importe quel point du programme. Des années d'expérience ont convaincu les programmeurs que les données devaient rester les plus locales possibles et que leur accès devait être restreint.

Q Dans un prototype de fonction, quand dois-je utiliser le mot-clé `inline` ?

R Lorsque la fonction n'excède pas deux instructions et qu'elle est peu appelée.

Q Pourquoi les modifications apportées aux valeurs des paramètres d'une fonction ne se reflètent-elles pas dans la fonction appelante ?

R Les paramètres sont passés par valeur à la fonction, ce qui signifie qu'ils sont dupliqués et que la valeur d'origine est conservée dans la fonction appelante. Pour en savoir plus, reportez-vous à la section "Principe des fonctions".

Q Si les paramètres sont passés par valeur, que faire si l'on souhaite que les modifications soient reflétées dans la fonction appelante ?

R La solution consiste à utiliser les pointeurs ou des références (voir les Chapitres 8 et 9). Vous y trouverez aussi une solution pour qu'une fonction puisse renvoyer plusieurs valeurs.

Q Je rencontre ces deux fonctions :

```
int Surface(int largeur, int longueur = 1);
int Surface(int taille);
```

Q Y a-t-il surcharge ? Nous avons un nombre différent de paramètres, mais le premier prototype a une valeur par défaut.

R Les déclarations seront compilées sans erreur, mais si vous invoquez la fonction `Surface` avec un seul paramètre, vous obtiendrez une erreur signalant qu'il y a ambiguïté entre `Surface(int, int)` et `Surface(int)`.

Testez vos connaissances

1. Quelle différence y a-t-il entre un prototype de fonction et une définition de fonction ?
2. Les noms de paramètres doivent-ils être identiques dans le prototype, la définition et l'appel à la fonction ?
3. Comment déclarer une fonction qui ne renvoie pas de valeur ?
4. En l'absence de valeur renvoyée, quel sera le type du résultat ?
5. Qu'est-ce qu'une variable locale ?
6. Qu'est-ce que la portée ?
7. Que signifie le mot "récursivité" ?
8. Quand doit-on utiliser des variables globales ?
9. Qu'est-ce que la surcharge de fonction ?

Exercices

1. Écrivez le prototype de la fonction `Perimetre()`, qui renvoie un entier long non signé et a deux paramètres entiers courts non signés.
2. Écrivez la définition de la fonction `Perimetre()`, décrite ci-dessus. Les deux paramètres correspondent à la longueur et à la largeur d'un rectangle. Rappel : le périmètre est égal à deux fois la largeur + deux fois la longueur.

3. CHERCHEZ L'ERREUR : cette fonction est boguée.

```
#include <iostream>
void maFct(unsigned short int x);
int main()
{
   unsigned short int x, y;
   y = maFct(int);
   std::cout << "x : " << x << " y : " << y << "\n";
   return 0;
}

void maFct(unsigned short int x)
{
  return (4*x);
}
```

4. CHERCHEZ L'ERREUR : cette fonction est également boguée.

```
#include <iostream>
int maFct(unsigned short int x);
int main()
{
   unsigned short int x, y;
   x = 7;
   y = maFct(x);
   std::cout << "x : " << x << " y : " << y << "\n";
   return 0;
}

int maFct(unsigned short int x);
{
  return (4*x);
}
```

5. Écrivez une fonction qui reçoit deux paramètres entiers courts non signés, divise le premier par le second et renvoie le résultat. Si le second nombre est égal à zéro, la division est impossible et la fonction doit renvoyer la valeur `-1`.
6. Écrivez un programme qui invite à entrer deux valeurs puis qui appelle la fonction définie à la question 5. Affichez le résultat ou un message d'erreur si la valeur renvoyée est égale à `-1`.
7. Écrivez un programme qui demande un nombre et un exposant. Créez une fonction récursive qui met le nombre à la puissance indiquée. Par exemple, si le nombre entré est 2 et l'exposant 4, la fonction doit renvoyer 16.



6

Programmation orientée objet

Au sommaire de ce chapitre

- Les classes et les objets
- La définition d'une classe et la création d'objets associés
- La notion de fonction membre et de donnée membre
- Le rôle des constructeurs

C++ est-il orienté objet ?

À une certaine époque, C – le prédécesseur de C++ – était le langage de programmation le plus populaire pour le développement de logiciels professionnels. C'est lui qui a été utilisé pour créer les systèmes d'exploitation (comme Unix), pour la programmation en temps réel (contrôle des machines, des périphériques et de l'électronique). Ce n'est que plus tard qu'il a été employé comme langage de programmation conventionnel. Il avait pour but de simplifier la programmation en ce qui concerne l'aspect matériel.

C a été conçu pour être un compromis entre les langages de programmation de haut niveau, comme COBOL, et l'assembleur, qui est extrêmement performant, mais difficile à utiliser. C mettait en œuvre la programmation "structurée", où les problèmes sont décomposés en unités plus petites qui peuvent être répétées (les *procédures*) et les données sont regroupées en paquets (appelés *structures*).

Or, les langages de recherche, comme Smalltalk et CLU, avaient déjà commencé à choisir une nouvelle orientation (l'orientation objet), qui associait les données dans des assemblages comme les structures, mais avec les fonctionnalités des procédures, le tout en une seule unité : l'objet.

Le monde qui nous entoure est peuplé d'objets : voitures, chiens, arbres, nuages, fleurs, etc. Chacun possède ses caractéristiques propres (rapide, amical, noir...). La plupart ont un comportement (ils roulent, aboient, poussent...). Généralement, nous ne faisons pas référence aux données d'un chien, ni à la façon de les manipuler : nous voyons un chien comme un élément faisant partie de notre univers (son apparence, ce qu'il fait). Cela vaut également pour tout objet du monde réel intégré dans le domaine de l'informatique.

Les programmes écrits en ce début de XXI^{e} siècle sont beaucoup plus complexes que ceux écrits à la fin du précédent. Les programmes créés à l'aide de langages procéduraux se révèlent difficile à maintenir et coûteux à modifier. Les interfaces utilisateur graphiques, Internet, la téléphonie numérique et sans fil, et toutes les nouvelles technologies ont considérablement accru la complexité des projets alors même que les attentes des utilisateurs augmentaient.

Les langages de programmation orientés objet constituent un outil précieux pour les défis du développement logiciel. Bien qu'il n'existe pas de solution idéale pour le développement de logiciels complexes, ces langages créent un lien puissant entre les structures de données et les méthodes qui les manipulent, se rapprochant ainsi de notre manière de penser (nous, les programmeurs et les clients). La communication et la qualité des logiciels s'améliorent en conséquence. On ne pense désormais plus en termes de structures de données et de fonctions permettant de les manipuler, mais en termes d'objets, comme s'il s'agissait de leurs équivalents dans le monde réel, qui apparaissent et agissent d'une certaine façon.

C++ a été conçu comme un pont entre la programmation orientée objet et le langage C. L'objectif était de fournir une conception orientée objet à une plate-forme de développement de logiciels professionnels rapide, avec un intérêt tout particulier pour les performances. Vous verrez plus loin comment C++ a atteint ces objectifs.

Création de nouveaux types

Les programmes ont généralement pour but de résoudre des problèmes de la vie courante, comme gérer la liste des employés ou simuler le fonctionnement d'un système de chauffage. Bien qu'il soit possible de résoudre des problèmes complexes uniquement avec des nombres et des caractères, il est bien plus pratique de pouvoir créer des représentations des objets que vous voulez traiter texte. En d'autres termes, il est plus facile de simuler le fonctionnement d'un système de chauffage si vous pouvez créer des variables qui représentent les pièces, les capteurs, les thermostats et les radiateurs. Plus les variables seront proches de la réalité, plus le programme sera simple à écrire.

Au cours des chapitres précédents, vous avez étudié un certain nombre de types de variables, dont les entiers non signés et les caractères. Le type d'une variable fournit des informations précieuses sur celle-ci. Si, par exemple, vous déclarez `Hauteur` et `Largeur` comme des entiers courts non signés (`unsigned short`), vous savez qu'elles pourront accueillir une valeur comprise entre 0 et 65 535, en supposant qu'un entier non signé soit codé sur deux octets sur votre plate-forme. Si vous essayez d'y copier autre chose, le programme détectera une erreur. Vous ne pouvez donc pas stocker votre nom dans ces variables et vous ne devriez même pas essayer de le faire.

Le type d'une variable nous renseigne sur :

- sa taille en mémoire ;
- les informations qu'elle peut contenir ;
- les actions qui peuvent s'y appliquer.

Dans des langages traditionnels comme le C, les types étaient intégrés au langage. En C++, le programmeur a la possibilité d'étendre le langage de base en créant les types dont il a besoin – chacun de ces nouveaux types peut avoir toutes les fonctionnalités et la puissance des types prédéfinis.

Inconvénients de la création de types avec `struct`

Le langage C permet de créer de nouveaux types en combinant des variables au sein de structures (`struct`). Ces structures peuvent être assimilées à un nouveau type de données *via* l'instruction `typedef`.

Cette fonctionnalité a toutefois quelques lacunes :

- Les `struct` et les fonctions qui les manipulent ne sont pas liées ; on ne retrouve ces fonctions qu'en lisant les fichiers d'en-tête des bibliothèques disponibles et en recherchant celles qui ont un paramètre du nouveau type.

- La coordination des activités des fonctions liées à un `struct` particulier est plus difficile car tout ce qui se trouve dans le `struct` peut être modifié à tout moment à un autre endroit du programme. Il n'y a aucun moyen de protéger la structure des interférences extérieures.
- Les opérateurs prédéfinis ne fonctionnent pas sur les `struct` ; il n'est pas possible d'additionner deux `struct` avec le signe plus (+), même si cela semblerait naturel (par exemple lorsque deux `struct` représentent des éléments textuels complexes que l'on souhaite concaténer ensemble).

Présentation des classes et des membres

La création d'un type passe, en C++, par la déclaration d'une classe. Une classe est simplement un ensemble de variables, souvent de types différents, associées à un ensemble de fonctions.

Une voiture, par exemple, est un ensemble constitué de cinq roues, d'un moteur, d'une carrosserie avec des portes, de sièges, etc. Avec votre auto, vous pouvez avancer, reculer, tourner, vous arrêter, vous garer, etc. Une classe permet d'*encapsuler* les différents éléments et les fonctions dans un unique ensemble appelé *objet*.

L'encapsulation de tous les éléments en une classe présente de nombreux avantages pour le programmeur. Tout est regroupé au même endroit, ce qui facilite les actions sur les données (accès, copie, etc.). Les clients de la classe (c'est-à-dire les parties du programme qui utilisent la classe) peuvent utiliser les objets sans se soucier ni de leur structure, ni de leur mode de fonctionnement.

Une classe peut contenir n'importe quelle combinaison de types de variables, mais également d'autres types de classes. Les variables d'une classe sont généralement appelées *variables membres* ou *données membres*. Une classe `Voiture`, par exemple, pourrait avoir des variables membres représentant les sièges, le type d'autoradio, les pneus, etc.

Une classe peut aussi contenir des fonctions appelées *fonctions membres*, ou *méthodes*. Les fonctions membres font partie de la classe au même titre que les variables membres. Elles déterminent les actions pouvant être réalisées par la classe.

Généralement, les fonctions membres de la classe manipulent ses variables membres. Par exemple, les méthodes de la classe `Voiture`, par exemple, pourraient inclure les fonctions `Demarrer()` et `Freiner()`. Une classe `Chat` pourrait contenir des données membres correspondant à l'âge et au poids de l'animal et ses méthodes pourraient inclure les fonctions `Dormir()`, `Miauler()` et `ChasserSouris()`.

Déclaration d'une classe

La déclaration d'une classe donne au compilateur des informations sur la classe. Pour déclarer une classe, vous devez utiliser le mot-clé `class` suivi du nom de la classe, d'une accolade ouvrante et de la liste des données membres et des méthodes associées. Pour clore la déclaration, utilisez une accolade fermante suivie d'un point-virgule. Voici par exemple comment déclarer la classe `Chat` :

```
class Chat
{
 public:
   unsigned int sonAge;
   unsigned int sonPoids;
   void Miauler();
};
```

Cette déclaration n'alloue pas de mémoire pour un chat. Elle décrit uniquement ce qu'est un `Chat`, quelles sont ses données membres (`sonAge` et `sonPoids`) et ce qu'il sait faire (`Miauler()`). Même si la mémoire n'est pas allouée, elle indique au compilateur la taille de la classe et l'espace mémoire nécessaire à chaque chat. Si un entier occupe 4 octets sur l'ordinateur, un objet `Chat` aura donc une taille de 8 octets : 4 pour `sonAge` et 4 pour `sonPoids`. `Miauler()` n'occupera que l'espace nécessaire aux stockage des informations permettant de connaître son emplacement en mémoire ; il s'agit en fait d'un pointeur vers une fonction qui peut occuper 4 octets sur une plate-forme 32 bits.

Conventions d'attribution de noms

L'attribution d'un nom à chaque variable, fonction membre et classe fait partie de la tâche d'un programmeur. Vous avez appris au Chapitre 3 que les noms de variables et de constantes devaient être significatifs. `Chat`, `Rectangle` et `Employe` sont de bons noms de classes, par exemple. `Miauler()` et `StopperMoteur()` sont également de bons noms de méthodes puisqu'ils indiquent clairement le rôle de chaque fonction. Nombre de programmeurs choisissent de préfixer le nom de leurs variables membres avec `son` (ou `its`) comme dans `sonAge`, `sonPoids` ou `saVitesse`. Cela permet de distinguez plus facilement les variables membres de celles qui ne le sont pas.

Il est également possible d'utiliser d'autres préfixes, comme `monAge`, `monPoids` ou `maVitesse`, ou tout simplement la lettre `"m"` (comme membre), avec éventuellement un caractère de soulignement (_), par exemple `mAge` ou `m_age`.

Certains choisissent de préfixer leur nom de classe avec une lettre particulière, comme `cChat` ou `cPersonne` et d'autres préfèrent n'utiliser que des majuscules ou que des minuscules. Dans ce livre, les noms de classes commenceront tous par une majuscule comme dans `Chat` ou `Personne`.

De même, de nombreux programmeurs débutent tous les noms de méthodes par une majuscule et n'utilisent que des minuscules pour les noms des variables membres. Les mots les composant sont séparés par le caractère de soulignement comme dans `Chasser_Souris`, ou simplement indiqués avec la première lettre en majuscule (`ChasserSouris` ou `DessinerCercle`).

Le plus important est de choisir un style et de l'appliquer dans tous les programmes. Avec le temps, votre style évoluera, non seulement pour les conventions de noms, mais également pour l'indentation, l'alignement des accolades et les styles de commentaires.



La plupart des sociétés de développement ont bien sûr adopté un "standard maison" pour régler les problèmes de style. Cela garantit que tous les développeurs pourront relire aisément le code de leurs collègues. Malheureusement, cela s'applique également aux sociétés qui développent des systèmes d'exploitation et des bibliothèques de classes réutilisables, ce qui signifie généralement que les programmes C++ doivent travailler simultanément avec plusieurs conventions de nommage.



Comme on l'a déjà indiqué, C++ différencie les majuscules des minuscules. Les noms de classes, fonctions et variables devront donc tous être choisis sur le même modèle. Cela vous épargnera d'avoir à rechercher comment le nom de la classe a été orthographié : `Rectangle`*,* `RECTANGLE` *ou* `rectangle` *?*

Définition d'un objet

Après avoir déclaré une classe, vous pouvez l'utiliser comme nouveau type pour déclarer des variables de ce type. Il s'agit d'une opération aussi simple que la définition d'une variable entière :

```
unsigned int PoidsBrut;      // définition d'un entier non signé
Chat Frisky;                 // définition d'un chat
```

Dans ce code, on définit la variable `PoidsBrut` de type entier non signé, ainsi que l'objet `Frisky` de la classe (ou du type) `Chat`.

Classes et objets

Vous ne dorlotez pas la définition d'un chat, vous dorlotez un chat particulier. Vous faites une différence entre un chat en général et celui qui gît langoureusement sur le divan de votre salon. De la même façon, C++ fait la différence entre la classe `Chat` en général et

chaque objet `Chat` individuel. `Frisky` est un objet de la classe `Chat`, comme `PoidsBrut` est une variable de type entier non signé.

Un *objet* est une instance particulière d'une classe.

Accès aux membres d'une classe

Après avoir défini un objet réel `Chat` (`Frisky`, par exemple), vous pouvez accéder à ses membres à l'aide de l'opérateur point (`.`). Pour affecter 5 à la variable membre `sonPoids` de `Frisky`, on utilise donc l'expression suivante :

```
Frisky.sonPoids = 5;
```

Pour appeler la fonction `Miauler()`, il suffit de taper :

```
Frisky.Miauler();
```

Pour utiliser une méthode de la classe, il suffit de l'appeler. Dans l'exemple ci-dessus, vous appelez la fonction `Miauler()` de `Frisky`.

Affectez des valeurs aux objets, pas aux classes

En C++, vous ne pouvez pas affecter des valeurs à des types. L'attribution de valeurs concerne les variables. Par exemple, vous n'avez pas le droit d'écrire :

```
int = 5;                    // erreur
```

Le compilateur détectera l'erreur, car il est impossible d'affecter 5 à un entier. En revanche, vous pouvez définir une variable de type entier et lui affecter la valeur 5. Exemple :

```
int  x;                     // définir x comme un entier
x = 5;                      // affecter la valeur 5 à x
```

L'expression suivante produit également une erreur :

```
Chat.sonAge = 5;              // erreur
```

Le compilateur produira une erreur, car il est impossible d'affecter la valeur 5 au membre `sonAge` de la classe `Chat`. Vous devez créer un objet `Chat` spécifique, puis lui attribuer cette valeur. Exemple :

```
Chat Frisky;                // même chose que int x;
Frisky.sonAge = 5;          // même chose que x = 5;
```

Les membres non déclarés n'existent pas

Essayez l'expérience suivante : montrez votre chat à un enfant de trois ans et dites-lui "il s'appelle Frisky et il connaît un tour de magie : Frisky, aboie !". L'enfant gloussera et vous répondra : "mais non, idiot, les chats n'aboient pas".

Si vous écrivez :

```
Chat Frisky;        // Création d'un chat nommé Frisky
Frisky.Aboyer();    // Demande à Frisky d'aboyer
```

Le compilateur vous répondra aussi " mais non, idiot, les chats n'aboient pas" (bien qu'il utilisera sûrement d'autres termes moins poétiques, comme `[531] Error: Member function Aboyer not found in class Chat`). Le compilateur sait que `Frisky` ne peut pas aboyer car la classe `Chat` ne possède pas de méthode `Aboyer()` et il ne saura pas non plus que `Frisky` peut miauler si vous n'avez pas défini de méthode `Miauler()`.

Faire	*Ne pas faire*
● Déclarer une classe à l'aide du mot-clé `class`. ● Utiliser l'opérateur point (`.`) pour accéder aux membres et aux méthodes de la classe.	● Confondre une déclaration avec une définition. Une déclaration décrit une classe, une définition réserve de la mémoire pour un objet. ● Confondre classe et objet. ● Affecter des valeurs à une classe. En revanche, vous pouvez attribuer des valeurs aux données membres d'un objet.

Accès privé et accès public

Une déclaration de classe fait également appel à des mots-clés supplémentaires. `private` et `public` font partie des plus importants.

Ces deux mots-clés sont utilisés avec les membres d'une classe, que ce soient des données ou des méthodes. Les membres privés ne peuvent être accédées que depuis les méthodes de la classe elle-même alors que les membres publics peuvent être accédés *via* n'importe quel objet de la classe. Cette distinction est très importante, mais souvent source de confusion chez le développeur débutant. Par défaut, tous les membres d'une classe sont privés.

Pour clarifier les choses, reprenons l'exemple décrit plus haut :

```
class Chat
{
```

```
  unsigned int sonAge;
  unsigned int sonPoids;
  void Miauler();
};
```

Dans cette déclaration, `sonAge`, `sonPoids` et `Miauler()` sont privés, car tous les membres d'une classe sont privés par défaut. À moins de préciser le contraire, ils sont privés. Si vous créez un programme et que vous exécutez les deux instructions suivantes dans `main()` :

```
int main()
{
   Chat Mistigri;
   Mistigri.sonAge = 5; // erreur ! impossible d'accéder à des
                        // données privées !
   ...
```

le compilateur produira une erreur. En effet, en laissant ces membres privés, vous indiquez au compilateur que l'on ne pourra accéder à `sonAge`, `sonPoids` et `Miauler()` qu'à partir des fonctions membres de la classe `Chat`. Pourtant, ici, vous avez tenté d'accéder au membre `sonAge` de `Mistigri` depuis l'extérieur d'une méthode de `Chat`. Le fait que `Mistigri` soit un objet `Chat` ne signifie pas que vous pouvez accéder à ses composants privés (même s'ils sont visibles dans la déclaration).

Les débutants en C++ sont souvent troublés par ce fonctionnement. Je vous entends presque hurler "Je viens de dire que Mistigri est un chat ! Pourquoi ne peut-il pas accéder à son propre âge ?". En fait, il le peut, mais pas vous ! Dans ses propres méthodes, `Mistigri` peut accéder à tous ses membres, publics ou privés. Le fait que vous ayez créé un `Chat` ne signifie pas que vous pouvez voir ou modifier ses parties privées.

Vous pouvez avoir accès aux données membres de la classe `Chat` en rendant publics certains des membres :

```
class Chat
{
public:
    unsigned int  sonAge;
    unsigned int  sonPoids;
    void Miauler();
};
```

Dans cette déclaration, `sonAge`, `sonPoids` et `Miauler()` sont désormais publics. `Mistigri.sonAge = 5`, de l'exemple précédent, peut donc être compilé sans problème.



Le mot-clé `public` s'applique à tous les membres de la déclaration jusqu'à ce que l'on rencontre le mot-clé `private` – et vice versa. Cela vous permet de déclarer facilement des parties de la classe comme privées ou publiques.

Dans le Listing 6.1, les membres de la classe `Chat` sont publics.

Listing 6.1 : Accès aux membres *public* d'une classe simple

```
 1:  // Déclaration d'une classe Chat et
 2:  // définition d'un objet de la classe
 3:
 4:  #include <iostream>
 5:
 6:  class Chat          // déclare la classe Chat
 7:  {
 8:  public:             // les membres qui suivent sont publics
 9:     int sonAge;      // variable membre
10:     int sonPoids;    // variable membre
11:  };                  // remarquez le point-virgule
12:
13:  int main()
14:  {
15:     Chat Frisky;
16:     Frisky.sonAge = 5;   // affecte 5 à la variable membre
17:     std::cout << "Frisky est un chat qui a " ;
18:     std::cout << Frisky.sonAge << " ans.\n";
19:     return 0;
20:  }
```

Ce programme produit le résultat suivant :

```
Frisky est un chat qui a 5 ans.
```

La ligne 6 contient le mot-clé `class`, indiquant que ce qui suit est une déclaration de classe. Ici, cette classe s'appelera `Chat`.

Le corps de la déclaration commence par une accolade ouvrante à la ligne 7 et se termine par une accolade fermante et un point-virgule à la ligne 11. À la ligne 8, le mot-clé `public` suivi de deux-points signifie que tout ce qui suit est public jusqu'à la rencontre du mot-clé `private` ou de la fin de la déclaration.

Les lignes 9 et 10 contiennent les déclarations des membres `sonAge` et `sonPoids`.

La fonction principale du programme commence à la ligne 13 et la ligne 15 définit `Frisky` comme une instance de `Chat` : c'est donc un objet de la classe `Chat`. L'âge est défini à la ligne 16, puis affiché. Notez comment on accède aux membres de l'objet `Frisky` aux

lignes 16 et 18 : `sonAge` est accédé à l'aide du nom d'objet (`Frisky` ici), suivi d'un point puis du nom du membre (`sonAge` ici).



Essayez de mettre en commentaire (c'est-à-dire de désactiver) la ligne 8, puis recompilez le programme. La ligne 16 produira une erreur car `sonAge` *n'est plus public mais est revenu à l'accès par défaut, comme les autres membres, et cet accès est privé.*

Rendre privées les données membres

Une règle générale de la conception est que les données membres d'une classe doivent rester privées. Bien entendu, si toutes les données membres deviennent privées, vous risquez de vous demander comment accéder aux informations sur la classe. Si `sonAge` est privé, par exemple, comment définir ou récupérer l'âge d'un objet `Chat` ?

Pour accéder aux données privées d'une classe, vous devez créer des fonctions publiques, appelées *méthodes d'accès*. Ces méthodes sont des fonctions membres qui peuvent être appelées à d'autres endroits de votre programme pour accéder aux données privées.

Une *méthode d'accès publique* est une fonction membre de la classe capable de lire (récupérer) la valeur d'une variable membre privée de la classe et/ou de lui affecter une valeur.

Pourquoi faire intervenir un niveau supplémentaire d'accès ? Pourquoi ajouter d'autres fonctions alors qu'il est plus aisé et plus simple d'utiliser directement des données ? Pourquoi passer par des méthodes d'accès ?

La réponse à ces questions est que les méthodes d'accès permettent de dissocier la façon dont les données sont *stockées* et la manière dont elles sont *utilisées*. En obligeant le code client à passer par des méthodes d'accès, le concepteur de la classe peut ensuite modifier le mode de stockage des données membres (et donc également les méthodes d'accès) sans que cela ne perturbe le code client.

Une fonction qui accède directement à l'élément `sonAge` pour connaître l'âge d'un chat devra être réécrite si vous décidez, plus tard, de modifier le mode de stockage de cette information. En obligeant la fonction cliente à appeler `LireAge()` pour obtenir l'âge de notre félin, celle-ci n'a pas besoin de connaître le type de la variable qui contient l'âge, ni de savoir comment il est calculé.

Il va sans dire que cette technique simplifie énormément la maintenance des programmes. En outre, le code a une durée de vie supérieure car d'éventuelles modifications de conception n'affecteront plus les programmes qui l'utilisent.

Les méthodes d'accès peuvent également contenir des traitements supplémentaires. S'il y a peu de chances, par exemple, que l'âge du chat soit supérieur à 100, ou que son poids

soit égal à 1 000, ces valeurs ne devraient donc pas être autorisées. Les méthodes d'accès peuvent mettre en place ces types de contraintes et réaliser bien d'autres opérations.

Le Listing 6.2 déclare la classe `Chat` qui comprend des données membres privées et des méthodes d'accès publiques. Attention : ce code n'est pas exécutable s'il est compilé.

Listing 6.2 : Exemples de méthodes d'accès

```
 1:  // Déclaration de la classe Chat
 2:  // Les données membres sont privées, les méthodes d'accès
 3:  // publiques permettent l'affectation et la récupération
3a: // des données privées
 4:  class Chat
 5:  {
 6:  public:
 7:     // méthodes d'accès publiques
 8:     unsigned int LireAge();
 9:     void DefAge(unsigned int Age);
10:
11:     unsigned int LirePoids();
12:     void DefPoids(unsigned int Poids);
13:
14:     // fonctions membres publiques
15:     void Miauler();
16:
17:     // données membres privées
18:  private:
19:     unsigned int  sonAge;
20:     unsigned int  sonPoids;
21:  };
```

Cette classe contient cinq méthodes publiques. Les lignes 8 et 9, d'une part, et 11 et 12, d'autre part, correspondent respectivement aux méthodes d'accès de `sonAge` et de `sonPoids`. Vous pouvez constater que la ligne 8 déclare une méthode pour obtenir l'âge et que la ligne 9 en déclare une permettant de le modifier. Les lignes 11 et 12 font de même pour le poids. Ces méthodes d'accès permettent donc d'initialiser les valeurs des variables membres et de les lire.

La ligne 15 déclare la fonction membre publique `Miauler()`. Il ne s'agit pas d'une fonction d'accès ; elle ne lit pas ou ne modifie pas une variable membre mais effectue une action pour la classe qui consiste, ici, à afficher "Miaou", comme vous le verrez dans le programme du Listing 6.3.

Les variables membres sont déclarées aux lignes 19 et 20.

Pour définir l'âge de Frisky, vous devez passer la valeur à la méthode `DefAge()` :

```
Chat  Frisky;
Frisky.DefAge(5);  // fixe l'âge à l'aide de la méthode d'accès
```

Plus loin, vous trouverez le code permettant de faire fonctionner `SetAge` et les autres méthodes.

Déclarer des méthodes ou des données privées permet au compilateur de détecter des erreurs de programmation avant qu'elles ne deviennent des bogues. Tout programmeur peut contourner les restrictions d'accès s'il le souhaite. Bjarne Stroustrup, l'inventeur de C++, a d'ailleurs lui-même déclaré que les mécanismes des contrôles des accès protégeaient des accidents, mais pas de la fraude (ARM, 1990).

Le mot-clé `class`

Sa syntaxe se présente ainsi :

```
class nom_de_la_classe
{
   // mots clés des contrôles d'accès
   // variables et méthodes membres
};
```

Le mot-clé `class` permet de déclarer de nouveaux types. Une classe est un ensemble de données membres, qui sont des variables de différents types, y compris d'autres classes. La classe contient également des fonctions (ou méthodes) qui permettent de manipuler les données de la classe ou d'effectuer un certain nombre de services pour celle-ci.

On peut comparer le processus de définition des objets du nouveau type à celui d'une variable. On indique le type (le nom de la classe) suivi du nom de la variable (l'objet). Pour avoir accès aux fonctions et aux membres de cet objet, utilisez l'opérateur point (`.`).

Les sections d'une classe peuvent être déclarées privées ou publiques en utilisant les mots-clés `private` ou `public`. Par défaut, le contrôle d'accès est de type privé. L'accès ne change pas tant qu'un nouveau mot-clé n'est pas précisé dans le code source. Les déclarations de classe se terminent par une accolade fermante et un point-virgule.

Exemple 1

```
class Chat
{
  public:
  unsigned int sonAge;
  unsigned int sonPoids;
  void Miauler();
};
Chat  Frisky;
Frisky.sonAge = 8;
Frisky.sonPoids = 9;
Frisky.Miauler();
```

Exemple 2

```
class Voiture
{
  public:  // les cinq membres suivants sont publics
  void Demarrer();
  void Accelerer();
  void Freiner();
  void DefAnnee(int annee);
  int LireAnnee();
private:        // le reste est privé
   int sonAnnee;
   char sonModele [255];
};             // fin de la déclaration

Voiture Star5;        // crée une instance de Voiture
int achat;            // variable locale entière
Star5.DefAnnee(84) ;  // voiture achetée en 84
achat = Star5.LireAnnee(); // extraction du millésime
Star5.Demarrer();     // appel de la méthode Demarrer
```

Faire	*Ne pas faire*
● Utiliser des méthodes d'accès publiques. ● Accéder à des variables membres privées depuis des fonctions membres dans une classe.	● Déclarer les variables membres comme publiques si ce n'est pas nécessaire. ● Essayer d'accéder à des variables membres privées de l'extérieur d'une classe.

Implémentations des méthodes de la classe

Comme vous venez de le voir, une fonction d'accès fournit une interface publique aux données membres privées d'une classe. Comme n'importe quelle méthode de la classe, elle doit être associé à du code. Son implémentation est appelée *définition de la fonction*.

Cette définition commence de la même manière que celle d'une fonction ordinaire. On indique d'abord le type du résultat de la fonction, ou `void` si elle ne renvoie rien. Il est suivi du nom de la classe, puis de deux caractères deux-points, du nom de la fonction et de ses paramètres. Le Listing 6.3 montre comment déclarer la classe `Chat` et implémenter ses méthodes d'accès et une méthode membre générale.

Listing 6.3 : Implémentation des méthodes d'une classe simple

```
1:  // Déclaration d'une classe et
2:  // définition des méthodes de la classe,
3:  #include <iostream>          // pour cout
```

```
 4:
 5:  class Chat        // début de la déclaration de la classe
 6:  {
 7:  public:                      // début de section publique
 8:     int LireAge();            // méthode d'accès
 9:     void DefAge (int age);    // méthode d'accès
10:     void Miauler();           // méthode générale
11:  private:                     // début de section privée
12:     int sonAge;               // variable membre
13:  };
14:
15:  // LireAge, méthode d'accès publique
16:  // renvoie la valeur du membre sonAge
17:  int Chat::LireAge()
18:  {
19:     return sonAge;
20:  }
21:
22:  // définition de DefAge,
23:  // fonction d'accès publique
24:  // définit le membre sonAge
25:  void Chat::DefAge(int age)
26:  {
27:     // initialise la variable membre sonAge avec
28:     // la valeur passée par le paramètre age
29:     sonAge = age;
30:  }
31:
32:  // définition de la méthode Miauler
33:  // renvoie : void
34:  // paramètres : Aucun
35:  // rôle : afficher "Miaou" à l'écran
36:  void Chat::Miauler()
37:  {
38:     std::cout << "Miaou.\n";
39:  }
40:
41:  // créer un objet Chat, définir son âge, le faire miauler,
42:  // dire son âge, le faire miauler de nouveau.
43:  int main()
44:  {
45:     Chat Frisky;
46:     Frisky.DefAge(5);
47:     Frisky.Miauler();
```

```
48:     std::cout << "Frisky est un chat qui a " ;
49:     std::cout << Frisky.LireAge() << " ans.\n";
50:     Frisky.Miauler();
51:     return 0;
52: }
```

Ce programme produit le résultat suivant :

```
Miaou.
Frisky est un chat qui a 5 ans.
Miaou.
```

La classe `Chat` est définie de la ligne 5 à la ligne 13. À la ligne 7, le mot-clé `public` demande au compilateur de créer un ensemble de membres publics. La ligne 8 déclare la méthode d'accès publique `LireAge()`, qui a pour but d'extraire la valeur de la variable membre `sonAge` déclarée à la ligne 12. À la ligne 9, la fonction d'accès publique `DefAge()` est suivie d'un paramètre entier qui sera affecté à la variable membre `sonAge`.

La ligne 10 déclare la méthode `Miauler()`. Il ne s'agit pas d'une méthode d'accès. Il s'agit plutôt d'une méthode générale qui affiche "Miaou" à l'écran.

La section `private` commence à la ligne 11 : elle comprend la déclaration de la variable membre `sonAge` à la ligne suivante. Enfin, la déclaration de classe se termine par une accolade fermante et un point-virgule à la ligne 13.

La fonction membre `LireAge()` est définie entre les lignes 17 et 20. Cette méthode ne prend pas de paramètre mais renvoie une valeur entière. Vous remarquerez, à la ligne 17, qu'elle contient le nom de la classe suivi de deux caractères deux-points et du nom de la fonction. Le compilateur peut ainsi comprendre que la fonction `LireAge()` que vous définissez ici est celle que vous avez déclarée dans la classe `Chat`. Hormis sa ligne d'en-tête, la fonction `LireAge()` est créée comme n'importe quelle fonction.

Cette fonction n'occupe qu'une seule ligne et renvoie la valeur de `sonAge`. Notez que la fonction `main()` n'a pas accès à la variable `sonAge` puisqu'elle est définie comme membre privé de la classe `Chat`. En fait, la fonction principale ne peut avoir accès à la variable `sonAge` qu'à travers la fonction publique `LireAge()` qui est une fonction membre de la classe `Chat`. Cela permet de connaître l'âge de Frisky.

La fonction membre `DefAge()` est définie à la ligne 25. On voit que cette fonction prend une valeur entière, appelée `age`, et qu'elle ne renvoie pas de résultat car son type est `void`. `SetAge()` prend la valeur du paramètre `age` et l'attribue à `sonAge` à la ligne 29. Cette fonction est membre de la classe `Chat` et, à ce titre, peut avoir accès à la variable privée `sonAge`.

La méthode `Miauler()` de la classe `Chat` est définie à la ligne 36. Elle affiche le mot "Miaou" à l'écran. Vous remarquerez la présence du caractère `\n` qui permet de passer à la

ligne suivante. Vous pouvez constater que `Miauler()` est définie comme les méthodes d'accès : elle commence par le type de son résultat, suivi du nom de la classe, du nom de la fonction et de ses paramètres (aucun dans ce cas).

Le programme commence véritablement à la ligne 43 avec la fameuse fonction `main()`. À la ligne 45, `main()` déclare un objet nommé `Frisky`, du type `Chat`. Dit autrement, `main()` déclare un `Chat` nommé `Frisky`. La ligne 46, affecte la valeur 5 à la variable membre `sonAge` par l'intermédiaire de la fonction d'accès `DefAge()`. Pour appeler cette méthode, le programme utilise le nom de l'objet (`Frisky`) suivi de l'opérateur de membre (`.`) et du nom de la méthode (`DefAge()`). Les autres méthodes de l'objet peuvent être appelées de la même façon.



Les termes fonction membre et méthode sont interchangeables.

La fonction membre `Miauler()` est appelée à la ligne 47. À la ligne suivante, l'âge du chat s'affiche par le biais de la fonction d'accès publique `LireAge()`. Enfin, la fonction `Miauler()` est appelée de nouveau à la ligne 50. Même si ces méthodes font partie d'une classe (`Chat`) et sont utilisées *via* un objet (`Frisky`), elles agissent comme les fonctions vues précédemment.

Ajout de constructeurs et de destructeurs

Pour définir une variable entière, vous pouvez procéder de deux façons. Vous pouvez définir la variable, puis lui affecter ultérieurement une valeur dans le programme. Exemple :

```
int Poids;              // définition d'une variable
...                     // autres instructions
Poids = 7;              // affectation d'une valeur
```

Il est également possible de définir une variable et de l'initialiser aussitôt. Exemple :

```
int Poids = 7;          // définition et initialisation à 7
```

L'initialisation consiste à définir une variable et à lui affecter une valeur. Bien entendu, vous pouvez modifier cette valeur ultérieurement, car l'initialisation n'est une opération ni irréversible ni définitive. Elle garantit que la variable aura toujours une valeur significative.

Comment initialiser les données membres d'une classe ? Vous pouvez le faire en utilisant une fonction membre spéciale appelée *constructeur*, qui peut prendre autant de paramètres

que nécessaire, mais qui ne renvoie pas de valeur (pas même `void`). En fait, le constructeur est une méthode de la classe qui porte le même nom que celle-ci.

Lorsque vous déclarez un constructeur, il est souhaitable de déclarer également un *destructeur*. Alors que les constructeurs créent et initialisent des objets de la classe, les destructeurs libèrent la mémoire ou les ressources qui ont été allouées (soit dans le constructeur, soit au cours de la vie de l'objet). Un destructeur porte toujours le nom de la classe précédé d'un tilde (~). Il ne prend pas de paramètre et ne renvoie pas de valeur. Si vous deviez déclarer un destructeur pour la classe `Chat`, sa déclaration serait donc la suivante :

```
~Chat();
```

Constructeurs et destructeurs par défaut

Il existe différents types de constructeurs ; certains acceptent des paramètres, d'autres non. Un constructeur sans paramètre est appelé *constructeur par défaut*. Il n'existe par contre qu'une sorte de destructeur qui, comme le constructeur par défaut, ne prend aucun paramètre.

Si vous ne créez ni constructeur ni destructeur, le compilateur vous en fournit automatiquement.

Ce constructeur et ce destructeur par défaut créés par le compilateur ne prennent pas de paramètres et, en fait, ne font rien. Si vous souhaitez qu'ils fassent quelque chose, vous devez créer vos propres constructeur ou destructeur par défaut.

Appeler le constructeur par défaut

À quoi peut bien servir un constructeur qui ne fait rien ? C'est en partie une question de forme. Tous les objets doivent être "construits" puis "détruits" et ces fonctions "inopérantes" sont appelées dans le cadre du processus de construction et de destruction.

Pour déclarer un objet sans passer de paramètres, comme ici :

```
Chat Felix;           // Felix ne prend pas de paramètres
```

vous devez avoir un constructeur de la forme :

```
Chat();
```

Lorsque vous définissez un objet d'une classe, le constructeur est appelé. Si le constructeur de `Chat` attend deux paramètres, la définition d'un objet `Chat` sera de la forme :

```
Chat Frisky(5, 7);
```

Dans cet exemple, le premier paramètre pourrait être son âge et le second son poids. Si le constructeur attend un seul paramètre, vous écrirez :

```
Chat Frisky(3);
```

Si le constructeur ne prend pas de paramètres du tout (il s'agit alors du constructeur *par défaut*), supprimez les parenthèses et écrivez :

```
Chat Frisky;
```

C'est donc une exception à la règle qui veut que toutes les fonctions doivent être suivies de parenthèses, même celles qui ne prennent pas de paramètres. Vous pouvez donc écrire :

```
Chat Frisky;
```

car cela sera interprété comme un appel au constructeur par défaut. Il ne comprend ni paramètres ni parenthèses.

Vous n'êtes pas obligé de vous contenter du constructeur par défaut produit par le compilateur : vous pouvez écrire le vôtre, c'est-à-dire un constructeur sans paramètres. Le corps de ce constructeur par défaut peut ainsi contenir du code vous permettant d'initialiser l'objet comme vous le souhaitez. En réalité, il est toujours conseillé de définir un constructeur et d'initialiser les variables membres avec des valeurs par défaut appropriées afin que l'objet se comporte toujours correctement.

Lorsque vous déclarez un constructeur, n'oubliez non plus pas de définir un destructeur même s'il ne fait rien. Bien que celui fourni par défaut par le compilateur fonctionnera correctement, il est préférable de le définir soi-même afin de rendre le code plus clair.

Le Listing 6.4 réécrit la classe `Chat` pour qu'elle utilise un constructeur qui n'est pas celui par défaut et qui se charge d'initialiser l'âge de l'objet avec la valeur qu'on lui a transmis. L'appel du destructeur est également clairement indiqué.

Listing 6.4 : Utilisation de constructeurs et de destructeurs

```
 1:  // Déclaration de constructeurs et de
 2:  // destructeurs pour la classe Chat
 3:  // Le programmeur crée un constructeur par défaut
 4:  #include <iostream>         // pour cout
 5:
 6:  class Chat       // début de la déclaration de la classe
 7:  {
 8:  public:                     // début de section publique
 9:     Chat(int initialAge);    // constructeur
10:     ~Chat();                 // destructeur
11:     int LireAge();           // fonction d'accès
12:     void DefAge(int age);    // fonction d'accès
```

```
13:     void Miauler();
14:  private:                    // début de section privée
15:     int sonAge;              // variable membre
16:  };
17:
18:  // constructeur de Chat,
19:  Chat::Chat(int initialAge)
20:  {
21:     sonAge = initialAge;
22:  }
23:
24:  Chat::~Chat()     // destructeur, ne fait rien
25:  {
26:  }
27:
28:  // LireAge, fonction d'accès publique
29:  // renvoie la valeur du membre sonAge
30:  int Chat::LireAge()
31:  {
32:     return sonAge;
33:  }
34:
35:  // Définition de DefAge,
36:  // fonction d'accès publique
37:  void Chat::DefAge(int age)
38:  {
39:     // affecter à la variable membre sonAge
40:     // la valeur passée par le paramètre age
41:     sonAge = age;
42:  }
43:
44:  // définition de la méthode Miauler
45:  // renvoie : void
46:  // paramètres : Aucun
47:  // rôle : imprime "Miaou" à l'écran
48:  void Chat::Miauler()
49:  {
50:     std::cout << "Miaou.\n";
51:  }
52:
53:  // Créer un objet chat, définir son âge, le faire miauler,
54:  // afficher son âge, le faire miauler de nouveau.
55:  int main()
56:  {
57:     Chat Frisky(5);
58:     Frisky.Miauler();
59:     std::cout << "Frisky est un chat qui a " ;
```

```
60:     std::cout << Frisky.LireAge() << " ans.\n";
61:     Frisky.Miauler();
62:     Frisky.DefAge(7);
63:     std::cout << "Maintenant Frisky a " ;
64:     std::cout << Frisky.LireAge() << " ans.\n";
65:     return 0;
66: }
```

Ce programme produit le résultat suivant :

```
Miaou.
Frisky est un chat qui a 5 ans.
Miaou.
Maintenant Frisky a 7 ans.
```

Le Listing 6.4 ressemble étrangement au listing précédent. À la ligne 9, pourtant, le constructeur prend en paramètre un entier, tandis que la ligne 10 déclare le destructeur, qui ne prend pas de paramètre. Les destructeurs ne prennent jamais de paramètres et, comme les constructeurs, ils ne renvoient aucune valeur (pas même `void`).

Les lignes 19 à 22 montrent l'implémentation du constructeur, qui ressemble à celle de la méthode d'accès `DefAge()`. Comme vous pouvez le constater, le nom de classe précède le nom du constructeur, ce qui identifie la méthode, `Chat()` ici, comme une partie de la classe `Chat`. C'est un constructeur, qui ne renvoie donc pas de valeur, même pas `void`. À la ligne 21, il prend toutefois une valeur en paramètre qui est affectée à la donnée membre `sonAge`.

Les lignes 24 à 26 décrivent l'implémentation du destructeur `~Chat()`. Pour l'instant, cette fonction ne fait rien mais vous devez inclure sa définition si vous l'avez déclarée dans la déclaration de la classe. Comme le constructeur et les autres méthodes, elle est également précédée du nom de la classe.

La ligne 57 définit l'objet `Chat Frisky` en passant la valeur 5 au constructeur de cet objet. Il est inutile d'appeler `DefAge()` car `Frisky` a été créé avec la valeur 5 qui a été affectée à la variable membre `sonAge` à la ligne 60. À la ligne 62, cette variable est modifiée pour valoir 7. C'est cette valeur qui s'affiche à l'écran deux lignes plus loin.

Faire	*Ne pas faire*
● Initialiser des objets à l'aide de constructeurs. ● Ajouter un destructeur si vous ajoutez un constructeur.	● Attribuer un type de résultat aux constructeurs et aux destructeurs. ● Attribuer des paramètres aux destructeurs.

Inclure des fonctions membres *const*

Vous avez déjà employé le mot-clé `const` pour déclarer des variables qui ne changeront pas. Vous pouvez également l'utiliser avec les fonctions membres d'une classe. Lorsque l'on déclare une méthode `const`, on s'engage à ce que cette méthode ne modifie jamais aucun membre de la classe.

Pour déclarer une méthode `const`, il suffit d'intercaler le mot-clé entre la liste des paramètres et le point-virgule qui termine la déclaration de la méthode. Par exemple :

```
void Fonction() const;
```

Cette ligne déclare une méthode membre constante appelée `Fonction()` qui ne prend pas de paramètre et qui ne renvoie rien. Vous savez qu'elle ne changera aucune des données membres de la même classe car elle a été déclarée `const`.

Les méthodes d'accès qui ne font que renvoyer des valeurs sont souvent déclarées constantes. Nous avons vu plus haut que la classe `Chat` contenait deux fonctions d'accès :

```
void DefAge(int Age);
int LireAge();
```

La fonction `DefAge()` ne peut pas être constante puisqu'elle modifie la variable membre `sonAge`. En revanche, `LireAge()` peut et devrait être constante car elle ne modifie pas du tout les objets de la classe : elle ne fait que renvoyer la valeur de `sonAge`. Il serait donc préférable de déclarer ces fonctions sous la forme suivante :

```
void DefAge(int Age);
int LireAge() const;
```

Si une fonction modifie la valeur de l'un des membres d'un objet alors qu'elle est déclarée comme constante, le compilateur produira une erreur. Si, par exemple, la fonction `LireAge()` devait mémoriser le nombre de fois où un chat a dû dire son âge, une erreur de compilation serait produite, puisque vous modifieriez alors l'objet `Chat` lorsque cette méthode serait appelée.

Les méthodes constantes sont très intéressantes puisqu'elles permettent au compilateur de détecter les erreurs avant l'exécution du programme. C'est pourquoi il est souhaitable d'en déclarer aussi souvent que possible dans les applications.

Interface et implémentation

On appelle *client* toute partie d'un programme qui crée et utilise les objets d'une classe. L'interface publique (la déclaration de la classe) peut être considérée comme le *contrat* qui la lie aux clients. Ce contrat précise le comportement de la classe.

La déclaration de la classe `Chat`, par exemple, indique que l'âge de chaque chat peut être initialisé dans son constructeur, modifié par sa fonction d'accès `DefAge()`, puis lu par sa fonction d'accès `LireAge()`. Vous promettez également que chaque chat saura miauler grâce à la fonction `Miauler()`. Notez que vous ne dites rien de la variable membre `sonAge` dans l'interface publique ; il s'agit d'un détail d'implémentation qui ne fait pas partie de votre contrat. Vous fournirez un âge (`LireAge()`) et le définirez (`DefAge()`), mais le mécanisme (`sonAge`) reste invisible.

Si vous définissez `LireAge()` comme une fonction constante, ce qui est souhaitable, le contrat stipule également que son appel ne modifiera pas l'objet `Chat` sur lequel elle est appelée.

C++ étant fortement typé, le compilateur fera respecter ces contrats en produisant une erreur à chaque fois que le programme tentera de les violer. Le Listing 6.5 montre un programme qui ne se compilera pas à cause de ses violations de contrat.



La compilation du Listing 6.5 échouera !

Listing 6.5 : Programme contenant plusieurs erreurs de compilation

```
1:  // Démonstration d'erreurs de compilation
2:  // La compilation échouera!
3:  #include <iostream>            // pour cout
4:
5:  class Chat
6:  {
7:  public:
8:     Chat(int initialAge);
9:     ~Chat();
10:    int LireAge() const;      // fonction d'accès constante
11:    void DefAge (int age);
12:    void Miauler();
13: private:
14:    int sonAge;
15: };
16:
17: // constructeur de Chat,
18: Chat::Chat(int initialAge)
19: {
20:    sonAge = initialAge;
21:    std::cout << "Constructeur de Chat \n";
22: }
23:
```

```
23a:  // destructeur, ne réalise aucune action
 24:  Chat::~Chat()
 25:  {
 26:     std::cout << "Destructeur de Chat \n";
 27:  }
 28:  // LireAge, fonction constante
 29:  //, mais il y a violation !
 30:  int Chat::LireAge() const
 31:  {
 32:     return (sonAge++);           // violation de const !
 33:  }
 34:
 35:  // définition de DefAge,
 36:  // fonction d'accès publique
 37:
 38:  void Chat::DefAge(int age)
 39:  {
 40:     // définir la variable membre sonAge à la
 41:     // valeur passée par le paramètre age
 42:     sonAge = age;
 43:  }
 44:
 45:  // définition de la méthode Miauler
 46:  // renvoie : void
 47:  // paramètres : Aucun
 48:  // rôle : Affiche "Miaou" à l'écran
 49:  void Chat::Miauler()
 50:  {
 51:     std::cout << "Miaou.\n";
 52:  }
 53:
 54:  // démonstration des différentes violations de
 55:  // l'interface, et erreurs de compilation correspondantes
 56:  int main()
 57:  {
 58:     Chat Frisky;       // ne correspond pas à la déclaration
 59:     Frisky.Miauler();
 60:     Frisky.Aboyer();      // la méthode n'est pas déclarée
 61:     Frisky.sonAge = 7;    // sonAge est privée
 62:     return 0;
 63:  }
```

Le programme ne produit pas de résultat puisqu'il n'est pas compilé.

La ligne 10 déclare `LireAge()` comme une fonction d'accès constante alors qu'elle incrémente la variable membre `sonAge` à la ligne 32. Le compilateur a détecté une erreur et l'a signalée.

`Miauler()` n'est pas déclarée comme une méthode constante en ligne 12, ce qui en soi ne constitue pas une erreur mais est une mauvaise habitude de la part du programmeur. En effet, cette fonction ne doit en rien affecter les variables membres de la classe `Chat`. C'est pourquoi il est préférable de la déclarer comme méthode constante.

L'objet `Frisky` de la classe `Chat` est créé à la ligne 58. Le constructeur associé comprend un paramètre entier, or on n'en passe aucun : le compilateur détecte une erreur.



Si vous fournissez ne serait-ce qu'un constructeur, le compilateur n'en créera pas. Cela signifie que si vous créez quelque part un constructeur avec un paramètre, vous n'aurez pas de constructeur par défaut automatique et vous devrez l'écrire vous-même.

La ligne 60 appelle la méthode `Aboyer()`. Or, celle-ci n'a pas été déclarée.

La variable `sonAge` reçoit la valeur 7 à la ligne 61, c'est-à-dire dans le corps du programme, ce qui constitue une erreur puisqu'il s'agit d'une donnée membre privée.

Pourquoi détecter des erreurs à l'aide du compilateur ?

Il est illusoire d'espérer écrire du code sans erreurs. Toutefois, il est possible d'en éviter un certain nombre, en les détectant et en les résolvant dès le début du processus.

Les erreurs de compilation sont la bête noire des programmeurs. Pourtant, leur détection révèle la qualité du compilateur. En effet, un langage faiblement typé permet d'effectuer des opérations non autorisées, ce qui produira des erreurs au moment de l'exécution. Pire encore, le test n'apportera que peu d'aide pour découvrir les erreurs, car il existe trop de possibilités dans les vrais programmes pour avoir l'espoir de les tester tous.

Les erreurs de compilation sont préférables aux erreurs d'exécution, car elles peuvent être plus aisément repérées et corrigées. On reconnaît la qualité d'un programmeur à sa capacité à développer des applications exemptes d'erreurs d'exécution. Vous n'êtes pas infaillible. C'est pourquoi le compilateur est chargé de vérifier la cohérence du code avant de créer le fichier exécutable.

Où placer les déclarations de classes et les définitions de méthodes ?

Toute fonction déclarée dans une classe doit posséder une définition, que l'on appelle également *implémentation de la fonction*. Comme pour les autres fonctions, la définition d'une méthode de classe comprend un en-tête de fonction et un corps de fonction.

La définition doit se trouver dans un fichier dont le chemin d'accès doit être connu du compilateur. La plupart des compilateurs exigent que ce fichier se termine par l'extension `.c` ou `.cpp` – c'est cette dernière qui a été adoptée dans ce livre. Pour connaître les extensions prises en charge par votre compilateur, reportez-vous à sa documentation technique.



De nombreux compilateurs considèrent que les fichiers dont les noms se terminent par `.c` sont des fichiers sources C et que ceux qui se terminent par `.cpp` sont des fichiers sources C++. Vous pouvez utiliser l'une ou l'autre de ces extension, mais l'utilisation de `.cpp` réduira les confusions possibles.

Il est possible de placer également la déclaration dans ce fichier, bien que cette pratique soit peu recommandée en POO. Par convention, la plupart des programmeurs écrivent la déclaration dans un fichier en-tête portant le même nom que le fichier source, mais suivi de l'extension `.h`, `.hp` ou `.hpp`. Dans cet ouvrage, nous avons opté pour la dernière. Au besoin, consultez la documentation de votre compilateur.

Par exemple, la classe `Chat` est déclarée dans le fichier en-tête `Chat.hpp` et les définition des méthodes se trouvent dans le fichier `Chat.cpp`. Pour associer le fichier en-tête au fichier `.cpp`, ajoutez l'instruction suivante au début de `Chat.cpp` :

```
#include "Chat.hpp"
```

Le compilateur insèrera alors le code du fichier `Chat.hpp` comme s'il avait été saisi dans le fichier source. Notez que certains compilateurs exigent que la casse des caractères soit la même dans l'instruction `#include` et dans le système de fichiers.

Pourquoi séparer le contenu de votre fichier `.hpp` et de votre fichier `.cpp` pour finalement les réunir ? Le plus souvent, les clients de votre classe ne se soucient pas des détails d'implémentation. La lecture du fichier d'en-tête leur fournit toutes les informations dont ils ont besoin. En outre, vous pouvez également inclure le même fichier `.hpp` dans plusieurs fichiers `.cpp`.



La déclaration d'une classe est appelée interface, parce qu'elle contient un certain nombre d'informations précieuses pour l'utilisateur. Il s'agit, entre autres, de la structure de la classe, du type des données utilisées et des noms des fonctions appelées. L'interface figure généralement dans un fichier `.hpp`, plus connu sous le nom de fichier en-tête.

La définition d'une fonction renseigne le compilateur sur le comportement de cette dernière. Elle est appelée implémentation de la méthode et figure dans un fichier `.cpp`. Par ailleurs, les détails de l'implémentation ne concernent que le créateur de la classe ; les clients de la classe (les parties du programme qui l'utilisent) n'ont pas besoin de connaître la façon dont les méthodes sont implémentées.

Implémentation en ligne

Comme les fonctions classiques, les méthodes des classes peuvent être déclarées *en ligne*. Dans ce cas, le mot-clé `inline` doit figurer avant le type du résultat. Voici un exemple, avec la fonction `LirePoids()` :

```
inline int Chat::LirePoids()
{
   return sonPoids;      // renvoie la donnée membre Poids
}
```

Il est également possible de définir une fonction dans la déclaration de la classe, ce qui la définit automatiquement en ligne. Exemple :

```
class Chat
{
  public:
    int LirePoids() const { return sonPoids; }    // en ligne
    void DefPoids(int unPoids);
};
```

Notez la syntaxe de la définition de `LirePoids()`. Le corps de la fonction en ligne commence immédiatement après la déclaration de la méthode ; il n'y a pas de point-virgule après les parenthèses. Comme pour n'importe quelle fonction, la définition est comprise entre une accolade ouvrante et une accolade fermante. Les tabulations et les espaces n'influant pas sur son déroulement, la classe pourrait être définie ainsi :

```
class Chat
{
  public:
    int LirePoids() const
    {
       return sonPoids;
    }                          // en ligne
    void DefPoids(int unPoids);
};
```

Les Listings 6.6 et 6.7 recréent la classe `Chat`, mais la déclaration se trouve désormais dans le fichier en-tête `Chat.hpp` et l'implémentation des fonctions dans le fichier `Chat.cpp`. Notez que les fonctions d'accès et la fonction `Miauler()` sont déclarées en ligne dans le Listing 6.7.

Listing 6.6 : Déclaration de la classe *Chat* dans le fichier *Chat.hpp*

```
 1:  #include <iostream>
 2:  class Chat
 3:  {
 4:  public:
 5:     Chat (int ageInitial);
 6:     ~Chat();
6a:     // les trois fonctions suivantes sont en ligne
 7:     int LireAge() const { return sonAge;}
 8:     void DefAge (int age) { sonAge = age;}
 9:     void Miauler() const  { std::cout << "Miaou.\n";}
10:  private:
11:     int sonAge;
12:  };
```

Listing 6.7 : Implémentation de la classe *Chat* dans le fichier *Chat.CPP*

```
 1:  // Démonstration de fonctions en ligne
 2:  // et inclusion de fichiers en-tête
 3:  // N'oubliez pas d'inclure le fichier en-tête !
 4:  #include "Chat.hpp"
 5:
 6:
 7:  Chat::Chat(int ageInitial)   // constructeur
 8:  {
 9:     sonAge = ageInitial;
10:  }
11:
12:  Chat::~Chat()      // destructeur, ne fait rien
13:  {
14:  }
15:
16:  // Créer un chat, définir son âge, le faire miauler,
17:  // afficher son âge, le faire miauler de nouveau.
18:  int main()
19:  {
20:     Chat Frisky(5);
21:     Frisky.Miauler();
22:     std::cout << "Frisky est un chat qui a " ;
23:     std::cout << Frisky.LireAge() << " ans.\n";
24:     Frisky.Miauler();
25:     Frisky.DefAge(7);
26:     std::cout << "Maintenant Frisky a " ;
27:     std::cout << Frisky.LireAge() << " ans.\n";
28:     return 0;
29:  }
```

Ce programme produit le résultat suivant :

```
Miaou.
Frisky est un chat qui a 5 ans.
Miaou.
Maintenant, Frisky a 7 ans.
```

Le code présenté dans les deux listings ci-dessus ressemble étrangement à celui du Listing 6.4 mais on a déclaré ici trois méthodes en ligne dans le fichier en-tête `Chat.hpp` (Listing 6.6).

La fonction `LireAge()` est déclarée et implémentée en ligne à la ligne 6 de `Chat.hpp`,. Aux lignes 7 et 8, les autres fonctions sont également déclarées en ligne. Leur comportement n'a cependant pas été modifié.

La ligne 4 de `Chat.cpp` (Listing 6.7) contient la directive `#include "Chat.hpp"` qui inclut le contenu du fichier `Chat.hpp` dans le fichier source. Le précompilateur va donc inclure le fichier `Chat.hpp` comme si son code avait été tapé à partir de la ligne 5 de ce programme.

Cette technique permet de répartir les déclarations et l'implémentation dans des fichiers séparés tout en les rendant disponibles pour le compilateur, ce qui est une pratique très courante en C++. Les déclarations de classe sont généralement enregistrées dans un fichier `.hpp` qui est ensuite inclus dans le fichier `.cpp` associé.

Les lignes 18 à 29 reproduisent la fonction `main()` du Listing 6.4, ce qui montre qu'avoir mis ces fonctions en ligne ne modifie pas leurs fonctionnement.

Classes imbriquées

Il n'est pas rare de créer une classe complexe en déclarant plusieurs classes simples et en les incluant dans la déclaration de la classe complexe. Vous pouvez, par exemple, déclarer une classe `Roue`, une classe `Moteur`, une classe `Transmission`, etc., puis les combiner dans la classe `Voiture`. Ce processus établit une relation "*a-un*" ou relation d'appartenance. Une voiture *a un* moteur, elle *a des* roues et elle *a une* transmission.

Prenons un deuxième exemple. Un rectangle est formé de lignes, chaque ligne étant définie par deux points. Dans l'espace, un point est identifié à l'aide d'une coordonnée `x` et d'une coordonnée `y`. Le Listing 6.8 contient la déclaration complète d'une classe `Rectangle`, enregistrée dans le fichier `Rectangle.hpp`. Un rectangle étant défini par quatre lignes tracées entre quatre points d'intersection et chaque point correspondant à une position dans un graphique, il faut déclarer en premier la classe `Point` destinée à recevoir les coordonnées `x` et `y` de chaque point. Les deux classes sont implémentées dans le Listing 6.9.

Listing 6.8 : Déclaration complète d'une classe

```
 1:  // début de Rectangle.hpp
 2:  #include <iostream>
 3:  class Point     // contient les coordonnées x,y
 4:  {
 5:  // pas de constructeur, on utilise celui par défaut
 6:  public:
 7:     void SetX(int x) { sonX = x; }
 8:     void SetY(int y) { sonY = y; }
 9:     int GetX()const { return sonX;}
10:     int GetY()const { return sonY;}
11:  private:
12:     int sonX;
13:     int sonY;
14:  };    // fin de la déclaration de la classe Point
15:
16:
17:  class  Rectangle
18:  {
19:  public:
20:     Rectangle (int haut, int gauche, int bas, int droite);
21:     ~Rectangle () {}
22:
23:     int GetHaut() const { return sonHaut; }
24:     int GetGauche() const { return saGauche; }
25:     int GetBas() const { return sonBas; }
26:     int GetDroite() const { return saDroite; }
27:
28:     Point  GetHautGauche() const { return sonHautGauche ; }
29:     Point  GetBasGauche() const { return sonBasGauche; }
30:     Point  GetHautDroite() const { return sonHautDroite; }
31:     Point  GetBasDroite() const { return sonBasDroite; }
32:
33:     void SetHautGauche (Point pt)  { sonHautGauche = pt; }
34:     void SetBasGauche (Point pt)  { sonBasGauche= pt; }
35:     void SetHautDroite (Point pt) { sonHautDroite= pt; }
36:     void SetBasDroite (Point pt) { sonBasDroite= pt; }
37:
38:     void SetHaut(int haut) { sonHaut = haut; }
39:     void SetGauche (int gauche) { saGauche = gauche; }
40:     void SetBas (int bas) { sonBas = bas; }
41:     void SetDroite(int droite) { saDroite = droite; }
42:
43:     int GetSurface() const;
44:
45:  private:
```

```
46:     Point  sonHautGauche;
47:     Point  sonHautDroite;
48:     Point  sonBasGauche;
49:     Point  sonBasDroite;
50:     int    sonHaut;
51:     int    saGauche;
52:     int    sonBas;
53:     int    saDroite;
54:  };
55:  // fin de Rectangle.hpp
```

Listing 6.9 : Rect.cpp

```
 1:  // début de Rect.cpp
 2:  #include "Rectangle.hpp"
 3:  Rectangle::Rectangle(int haut, int gauche, int bas,
 3a:                      int droite)
 4:  {
 5:     sonHaut = haut;
 6:     saGauche = gauche;
 7:     sonBas = bas;
 8:     sonBas = droite;
 9:
10:     sonHautGauche.SetX(gauche);
11:     sonHautGauche.SetY(haut);
12:
13:     sonHautDroite.SetX(droite);
14:     sonHautDroite.SetY(haut);
15:
16:     sonBasGauche.SetX(gauche);
17:     sonBasGauche.SetY(bas);
18:
19:     sonBasDroite.SetX(droite);
20:     sonBasDroite.SetY(bas);
21:  }
22:
23:
24:  // calculer la surface du rectangle en trouvant les angles,
25:  // déterminer la longueur et la largeur et multiplier
26:  int Rectangle::GetSurface() const
27:  {
28:     int Largeur = saDroite - saGauche;
29:     int Hauteur = sonHaut - sonBas;
30:     return (Largeur * Hauteur);
31:  }
32:
33:  int main()
```

```
34:  {
35:     // initialiser une variable locale Rectangle
36:     Rectangle MonRectangle (100, 20, 50, 80 );
37:
38:     int Surface = MonRectangle.GetSurface();
39:
40:     std::cout << "Surface : " << Surface << "\n";
41:     std::cout << "Coordonnée X coin supérieur gauche : ";
42:     std::cout << MonRectangle.GetHautGauche().GetX();
43:     return 0;
44:  }
```

Ce programme produit le résultat suivant :

```
Surface : 3000
Coordonnée X coin supérieur gauche : 20
```

Les lignes 3 à 14 de `Rectangle.hpp` (Listing 6.8) déclarent la classe `Point` ; c'est elle qui va recevoir les coordonnées `x` et `y`. Comme vous pouvez le constater, ce programme ne fait guère appel à des objets `Point` ; cependant, ceux-ci peuvent servir à d'autres méthodes de tracé.



Certains compilateurs signalent une erreur si vous déclarez une classe appelée `Rectangle`. Ceci est généralement dû à l'existence d'une classe interne nommée `Rectangle`. Dans ce cas, renommez simplement la classe `monRectangle`, par exemple

Les variables membres `sonX` et `sonY` sont définies à la ligne 12 et à la ligne 13. À mesure que la valeur de la coordonnée `x` augmente, on se déplace vers la droite et on se déplace vers le haut lorsque la valeur de la coordonnée `y` augmente. Bien entendu, il est possible d'utiliser un autre système : dans certains programmes de fenêtrage, la fenêtre se déplace vers le bas à mesure que la coordonnée `y` augmente, par exemple.

Pour lire et définir les points `x` et `y`, la classe `Point` fait appel aux fonctions d'accès en ligne déclarées aux lignes 7 à 10. En outre, elle utilise le constructeur et le destructeur par défaut, ce qui signifie que vous devez définir les coordonnées explicitement.

La ligne 17 commence la déclaration de la classe `Rectangle`. Les quatre angles du rectangle sont alors définis.

Le constructeur de `Rectangle` prend en charge quatre entiers appelés respectivement `haut`, `gauche`, `bas` et `droite`. Ces paramètres sont affectés aux quatre variables membres (voir le Listing 6.9) afin de définir le rectangle.

Outre les fonctions d'accès décrites plus haut, la classe `Rectangle` possède une fonction `GetSurface()` (voir ligne 43). La surface n'est pas stockée dans une variable, mais calculée aux lignes 28 et 29 du Listing 6.9. Pour cela, la fonction `GetSurface()` détermine la hauteur et la largeur du rectangle, puis multiplie ces deux valeurs.

Pour extraire la coordonnée `x` de l'angle supérieur gauche du rectangle, vous devez lire la valeur `x` du point `HautGauche`. La fonction `GetHautGauche()` étant une méthode de `Rectangle`, elle a accès directement aux données privées, parmi lesquelles `sonHautGauche`. Cette valeur étant un `Point` et la valeur de `sonX` étant privée, elle ne peut pas y accéder directement et doit faire appel à la fonction d'accès publique `GetX()` pour obtenir cette coordonnée.

Le programme commence réellement à la ligne 33 du Listing 6.9,. Vous pouvez constater que la mémoire n'est pas allouée avant la ligne 36. Pour le moment, le compilateur sait créer un point et un rectangle, au cas où il en aurait besoin.

À la ligne 36, on définit un rectangle en passant les valeurs pour les quatre paramètres `haut`, `gauche`, `bas` et `droite` de son constructeur.

À la ligne 38, le programme crée la variable locale `Surface`, de type `int`. Elle reçoit la valeur de la surface, renvoyée par la fonction `GetSurface()` de la classe `Rectangle`. Un client de la classe `Rectangle` pourrait créer un objet `Rectangle` et déterminer sa surface sans connaître l'implémentation de la méthode `LireSurface()`.

Le fichier `Rectangle.hpp` est présenté dans le Listing 6.8. En examinant simplement la déclaration de la classe `Rectangle`, le programmeur sait que `GetSurface()` renvoie un entier. Le calcul de la surface ne concerne pas l'utilisateur de cette classe. En fait, l'auteur de `Rectangle` pourrait modifier le corps de la méthode `GetSurface()` sans influer sur les programmes qui utilise cette classe, du moment qu'elle renvoie un entier.

La ligne 42 du Listing 6.9 peut sembler étrange, mais réfléchissez à ce qui se passe. Cette ligne de code fournit les coordonnées *x* du point supérieur gauche de votre rectangle. Dans cette ligne, vous appelez la méthode `GetHautGauche()` de votre rectangle, qui vous renvoie à un `Point`. Vous voulez obtenir les coordonnées *x* à partir de ce `Point`. Vous avez vu que la méthode d'accès pour une coordonnée *x* dans la classe `Point` s'appelle `GetX()`. La ligne 42 rassemble donc simplement les méthodes d'accès `GetHautGauche()` et `GetX()` :

```
MonRectangle.GethautGauche().GetX();
```

Cet appel permet d'obtenir la coordonnée *x* du point supérieur gauche de l'objet `MonRectangle`.

Quelle est la différence entre une déclaration et une définition ?

Une déclaration introduit un nom, mais ne réserve pas d'emplacement en mémoire. Une définition alloue la mémoire correspondante.

À quelques rares exceptions près, toutes les déclarations sont aussi des définitions. Les exceptions les plus notables sont la déclaration d'une fonction globale (un prototype) et la déclaration d'une classe (généralement dans un fichier en-tête).

Les structures

Un proche cousin du mot-clé `class` est le mot-clé `struct` qui permet de déclarer une structure. En C++, une structure est une classe dont tous les membres sont publics par défaut. Une déclaration de structure comprend des données membres et des fonctions, comme une déclaration de classe. En fait, si vous respectez les bonnes habitudes de programmation en déclarant toujours explicitement chaque section de classe à l'aide des mots-clés `public` et `private`, il n'y aura aucune différence entre les deux.

Modifiez le Listing 6.8 :

- À la ligne 3, remplacez `class Point` par `struct Point`.
- À la ligne 17, remplacez `class Rectangle` par `struct Rectangle`.

Lancez à nouveau le programme et comparez les résultats obtenus. Il ne devrait pas y avoir de différence.

Vous vous demandez alors peut-être pourquoi il existe deux mots clés pour la même chose. Il s'agit d'un accident de l'histoire. C++ a été construit comme une extension du langage C, qui dispose de structures, bien qu'elles ne puissent pas contenir de méthodes. Bjarne Stroustrup, le créateur de C++, a utilisé le mot-clé `class` pour désigner ces structures étendues et la nouvelle visibilité par défaut de leurs membres. Cela a également permis de continuer à utiliser la vaste bibliothèque des fonctions C dans les programmes C++.

Faire	*Ne pas faire*
● Déclarer la classe dans un fichier en-tête (`.hpp`) et les fonctions membres dans le fichier source du programme (`.cpp`). ● Utiliser le mot-clé `const` le plus souvent possible.	● Poursuivre sans avoir compris le fonctionnement des classes.

Questions-réponses

Q Quelle est la taille d'un objet de la classe ?

R Elle est déterminée par la somme des tailles des différentes variables membres.Les méthodes n'occupent qu'un petit espace mémoire, utilisé pour conserver des informations sur l'emplacement de la méthode (un pointeur).

Certains compilateurs alignent les variables en mémoire de sorte que des variables de deux octets consomment en réalité un espace supérieur. Pour en savoir plus sur la gestion de la mémoire, reportez-vous au manuel utilisateur de votre compilateur.

Q Si je déclare une classe `Chat` comprenant un membre privé `sonAge`, puis que je définis deux objets de cette classe, `Frisky` et `Mistigri`. `Mistigri` a-t-il accès à la variable membre `sonAge` de `Frisky` ?

R Les différentes instances de la classe peuvent accéder aux données non publiques de l'autre, pas à leurs membres privés. `Frisky` et `Mistrigri` étant toutes les deux des instances de `Chat`, les fonctions membres de `Frisky` peuvent accéder aux données de `Frisky`, mais pas à celles de `Mistigri`.

Q Pourquoi ne pas rendre toutes les données membres publiques ?

R Les données membres privées sont propres à une classe, ce qui rend leur traitement et leur stockage totalement transparents pour les clients. Par exemple, les clients de la classe `Chat` n'ont pas accès directement à l'implémentation de la méthode `LireAge()`. Peu importe le traitement effectué, l'âge du chat sera récupéré. Le programmeur de la classe `Chat` peut changer ensuite son implémentation sans obliger tous les utilisateurs à modifier aussi leurs programmes.

Q Si l'utilisation d'une fonction `const` pour modifier une classe produit une erreur de compilation. Pourquoi ne pas supprimer ce mot-clé pour éviter des erreurs ?

R Si votre fonction membre ne doit logiquement pas modifier la classe à laquelle elle appartient, la présence du mot-clé `const` permet de détecter des erreurs. Par exemple, `LireAge()` n'est pas supposée modifier la classe `Chat`. Or si l'implémentation contient la ligne suivante :

```
if (sonAge = 100) cout << "Tiens, vous êtes centenaire !\n";
```

et que vous définissez la fonction `LireAge()` comme constante, le compilateur détectera l'erreur. En effet, l'instruction `if` était censée vérifier si `sonAge` était égal à 100 alors que vous affectez en réalité 100 à cette variable. Le compilateur a trouvé l'erreur, car vous tentez de modifier une méthode constante.

Une erreur de ce type est difficile à trouver en parcourant simplement les nombreuses lignes d'un fichier source. Le programme peut même sembler tourner correctement alors que `sonAge` a reçu une valeur incorrecte. Cela posera un problème tôt ou tard.

Q L'utilisation des structures est-elle encore nécessaire en C++ ?

R Certains programmeurs C++ réservent le mot-clé `struct` aux classes dépourvues de fonctions. Cette habitude, héritée du langage C, est à proscrire car une structures définie aujourd'hui peut très bien nécessiter l'ajout ultérieur de méthodes ; vous devriez alors modifier cette `struct` en `class` ou violer la règle que vous vous êtes fixé. La seule bonne raison d'utiliser une structure est lorsque vous devez appeler une fonction C existante qui exige un `struct` particulier.

Q Certaines personnes travaillant avec la programmation orientée objet utilisent le terme "instanciation". Que signifie-t-il ?

R L'instanciation n'est qu'un mot à la mode décrivant la création d'un objet à partir d'une classe. Un objet spécifique défini comme étant du type d'une classe constitue une *instance* de cette classe.

Testez vos connaissances

1. Qu'est-ce que l'opérateur point (`.`) ? À quoi sert-il ?
2. Qui réserve de l'espace en mémoire vive, la déclaration ou la définition ?
3. La déclaration d'une classe correspond-elle à son interface ou à son implémentation ?
4. Quelle est la différence entre des données membres privées et des données membres publiques ?
5. Les fonctions membres peuvent-elles être privées ?
6. Les données membres peuvent-elles être publiques ?
7. Si vous déclarez deux objets `Chat`, leurs données membres `sonAge` peuvent-elles avoir des contenus différents ?
8. Les déclarations de classe se terminent-elles par un point-virgule ? Et les définitions de méthodes de la classe ?
9. Quel serait l'en-tête de la fonction `Miaou` appartenant à la classe `Chat`, ne prenant en charge aucun paramètre et renvoyant `void` ?
10. Quelle est la fonction qui permet d'initialiser une classe ?

Exercices

1. Écrivez la déclaration de la classe `Personnel` contenant les données membres `sonAge`, `sonAnciennete` et `sonSalaire`.
2. Modifiez la déclaration de la classe `Personnel` de sorte que ses données membres soient privées et que des méthodes d'accès puissent lire et mettre à jour chacune d'elles.
3. Écrivez un programme qui crée deux objets de la classe `Personnel`. Pour chacun, définissez l'âge, l'ancienneté et le salaire, puis affichez ces valeurs. Vous devrez aussi ajouter le code des méthodes d'accès.
4. En vous inspirant de l'exercice précédent, écrivez le code d'une méthode qui affiche le salaire de chaque employé, arrondi à la centaine d'euros la plus proche.
5. Modifiez de nouveau la classe `Personnel` de façon à initialiser les données membres `sonAge`, `sonAnciennete` et `sonSalaire` lors de la création d'un objet.
6. CHERCHEZ L'ERREUR : pourquoi cette déclaration est-elle erronée ?

```
class Carre
{
public:
    int Cote;
}
```

7. CHERCHEZ L'ERREUR : pourquoi cette déclaration de classe n'est-elle pas très utile ?

```
class Chat
{
    int GetAge() const;
private:
    int sonAge;
};
```

8. CHERCHEZ L'ERREUR : il y a trois erreurs dans ce code. Où se cachent-elles ?

```
class  TV
{
public:
    void SetStation(int Station);
    int GetStation() const;
private:
    int saStation;
};
```

```
int main()
{
    TV maTele;
    maTele.saStation = 9;
    TV.SetStation(10);
    TV monAutreTele(2);
}
```



7

Déroulement d'un programme

Au sommaire de ce chapitre

- La nature et le rôle des boucles
- La création des différentes boucles
- Une alternative aux instructions imbriquées `if…else`

Un programme repose principalement sur des branchements et des boucles. Au Chapitre 4, vous avez appris comment le programme effectue un branchement à l'aide de l'instruction `if`.

Les boucles

En programmation, nombreux sont les traitements répétitifs, c'est-à-dire qui traitent plusieurs fois les mêmes groupes de données. Ils peuvent s'exécuter par récursivité (présentée au Chapitre 5) ou par itération. L'itération consiste à répéter plusieurs fois des instructions pour traiter des informations. La principale méthode d'itération est la *boucle*.

Les origines des boucles : *goto*

Aux débuts de l'informatique, les programmes manquaient cruellement de convivialité. Ils étaient courts et directs. Les boucles étaient formées d'une étiquette, de quelques instructions puis d'un saut vers cette étiquette.

En C++, une étiquette est simplement un nom suivi du signe deux-points (`:`) placé à gauche d'une instruction. Pour réaliser un saut, il suffit de taper le mot-clé `goto` suivi du nom de l'étiquette, comme le montre le Listing 7.1.

Listing 7.1 : Boucle à l'aide du mot-clé *goto*

```
 1:  // Listing 7.1
 2:  // Utilisation de goto
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:        using namespace std;
 8:        int compteur = 0;        // initialiser le compteur
 9:  boucle:
10:        compteur++;              // début de la boucle
11:        cout << "compteur : " << compteur << endl;
12:        if (compteur < 5)        // tester le compteur
13:           goto boucle;          // revenir au début
14:
15:        cout << "Fin du traitement. Valeur du compteur : "
15a:                << compteur << endl;
16:        return 0;
17:  }
```

Ce programme produit le résultat suivant :

```
compteur : 1
compteur : 2
compteur : 3
compteur : 4
compteur : 5
Fin du traitement. Valeur du compteur : 5
```

La ligne 8 initialise le compteur à zéro. Une étiquette appelée `boucle` marque le début de la boucle (ligne 9). Le compteur est incrémenté et sa nouvelle valeur apparaît à l'écran en ligne 11. La ligne 12 teste la valeur du compteur : si elle est inférieure à 5, la condition de l'instruction `if` est vérifiée et l'instruction `goto` s'exécute. Le programme revient sur l'étiquette `boucle` de la ligne 9 et l'itération continue jusqu'à ce que le compteur soit

égal à 5. En ce cas, le programme quitte la boucle et exécute la ligne 13 qui indique que le traitement est terminé.

Pourquoi jeter l'instruction *goto* aux oubliettes ?

Cette instruction est décriée, à juste titre, par les développeurs. Elle permet d'effectuer un saut vers n'importe quelle instruction située en amont ou en aval de la séquence en cours. *A priori*, cette fonctionnalité est très pratique mais, en réalité, lorsqu'elle est utilisée abusivement, elle rend les programmes illisibles et impossibles à maintenir car les sauts s'effectuent dans toutes les directions : on parle alors de *code spaghetti*.

L'instruction `goto`

Elle doit être suivie du nom de l'étiquette et elle provoque un saut inconditionnel vers cette dernière.

Exemple

```
if (valeur > 10)
   goto fin;
if (valeur < 10)
   goto fin;
cout << "valeur = 10 !";
fin:
   cout << "Fin";
```

D'autres instructions de boucle plus sophistiquées permettent d'éviter l'usage de `goto` : `for`, `while` et `do...while`.

Les boucles *while*

Une boucle `while` répète une séquence d'instructions *tant que* la condition de départ est vraie. Dans l'exemple du Listing 7.1, le compteur était incrémenté jusqu'à ce qu'il soit égal à 5. Pour cela, nous utilisions l'instruction `goto`. Au Listing 7.2, le même programme a été optimisé, à l'aide de l'instruction `while`.

Listing 7.2 : Boucles *while*

```
1:  // Listing 7.2
2:  // Une boucle while
3:  #include <iostream>
4:
```

```
 5:  int main()
 6:  {
 7:    using namespace std;
 8:    int compteur = 0;           // initialiser la condition
 9:
10:    while(compteur < 5)         // condition toujours vraie ?
11:    {
12:       compteur++;              // corps de la boucle
13:       cout << "Compteur : " << compteur << endl;
14:     }
15:
16:     cout << "Terminé. Le compteur vaut : "
16a:             << compteur << endl;
17:     return 0;
18:  }
```

Ce programme produit le résultat suivant :

```
Compteur : 0
Compteur : 1
Compteur : 2
Compteur : 3
Compteur : 4
Compteur : 5
Terminé. Le compteur vaut : 5
```

Ce programme illustre le fonctionnement d'une boucle `while`. La ligne 8 initialise à zéro la variable entière `compteur` qui est ensuite employée dans la condition de la boucle. La condition est testée : si elle est vraie, l'instruction figurant dans le corps de l'itération s'exécute. Dans cet exemple, le test est effectué à la ligne 10. Si la valeur du compteur est inférieure à 5, l'incrémentation a lieu à la ligne 12 ; le résultat est affiché à la ligne suivante. Dans le cas contraire, le corps de la boucle (lignes 11 à 14) n'est pas exécuté. Le programme se branche directement sur la ligne 15.

Notez qu'il vaut toujours mieux utiliser des accolades autour du bloc exécuté par une boucle, même lorsqu'il ne contient qu'une ligne de code. Cela permet d'éviter l'erreur classique consistant à placer par inadvertance un point-virgule juste après la condition, ce qui crée une boucle sans fin, comme ici :

```
int compteur = 0;
while ( compteur < 5 );
     compteur++;
```

Dans cet exemple, `compteur++` n'est jamais exécuté.

L'instruction `while`

Syntaxe de l'instruction `while` :

```
while ( condition )
  instruction;
```

`condition` est une expression C++ tandis que *`instruction`* correspond à une instruction ou à un bloc d'instructions C++ valide. Lorsque la condition est vérifiée, l'instruction s'exécute, puis la condition est de nouveau testée. Ce traitement se répète jusqu'à ce que la condition soit fausse. La boucle `while` prend alors fin et le programme lit l'instruction située immédiatement après l'instruction ou le bloc d'instructions. Exemple :

```
// incrémentation du compteur jusqu'à 10
int x = 0;
while (x < 10)
   cout << "X : " << x++;
```

Instructions *while* complexes

La condition testée dans une boucle `while` peut être aussi complexe que n'importe quelle expression C++ : elle peut donc contenir des opérateurs logiques comme `&&` (ET), `||` (OU) et ! (NON), comme le montre le Listing 7.3.

Listing 7.3 : Boucles *while* complexes

```
 1:  // Listing 7.3
 2:  // Instructions while complexes
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     using namespace std;
 8:     unsigned short petit;
 9:     unsigned long grand;
10:     const unsigned short MAXPETIT = 65535;
11:
12:     cout << "Entrez un petit nombre : ";
13:     cin >> petit;
14:     cout << "Entrez un grand nombre : ";
15:     cin >> grand;
16:
17:     cout << "Petit nombre : " << petit << "...";
18:
19:     // pour chaque itération, tester deux conditions
```

```
20:     while (petit < grand && petit < MAXPETIT)
21:     {
22:        // écrire un point à chaque incrément de 5000
22a:       if (petit % 5000 == 0)
23:           cout << ".";
24:
25:        petit++;
26:        grand -= 2;
27:     }
28:
29:     cout << "\nPetit nombre : " << petit <<
29a:            ", Grand nombre : " << grand << endl;
30:     return 0;
31:  }
```

Ce programme produit le résultat suivant :

```
Entrez un petit nombre : 2
Entrez un grand nombre : 100000
Petit nombre : 2.........
Petit nombre : 33335, Grand nombre : 33334
```

Il s'agit d'un jeu. Entrez deux nombres, le premier étant inférieur au second. Le programme incrémente de un point le petit nombre et décrémente de deux points le grand nombre. L'objectif est de déterminer le moment auquel les deux valeurs se rejoignent.

La saisie a lieu de la ligne 12 à la ligne 15. La ligne 20 définit une boucle `while` qui s'exécute tant que les deux conditions suivantes sont vraies :

- `petit < grand` (le petit nombre est inférieur au grand).
- `petit` ne dépasse pas la taille maximale autorisée pour un entier court (`MAXPETIT`).

La ligne 22 calcule la valeur de `petit` modulo 5 000. Ce calcul ne modifie pas la variable, mais extrait simplement le reste de la division. S'il est égal à 0, `petit` est un multiple de 5 000 et un point (`.`) apparaît à l'écran. La ligne 25 incrémente `petit` et la ligne suivante décrémente deux fois `grand`.

Si l'une des deux conditions est fausse, la boucle `while` prend fin et le programme se poursuit après l'accolade fermant la boucle à la ligne 27.



Pour en savoir plus sur l'opérateur modulo (`%`) et les opérateurs logiques, reportez-vous au Chapitre 3.

Les instructions *continue* et *break*

Dans certains cas, il est souhaitable de retourner au début de la boucle avant que le bloc d'instructions du corps de la boucle ne soit totalement exécuté. L'instruction `continue` permet d'effectuer cette opération.

Inversement, l'instruction `break` permet de sortir de la boucle et d'exécuter l'instruction située immédiatement après l'accolade fermante.

Nous utilisons ces instructions dans le Listing 7.4 car le jeu s'est compliqué. L'utilisateur saisit maintenant un petit nombre, un grand nombre, une valeur de saut et une valeur cible. Comme dans le listing précédent, le petit nombre est incrémenté de un point, alors que le grand nombre est décrémenté de deux points. La décrémentation ne s'effectuera pas lorsque le petit nombre est un multiple de la valeur du saut. Le jeu s'arrête lorsque `petit` est supérieur à `grand`. Si le grand nombre est égal à la valeur cible, un message apparaît à l'écran et le programme se termine.

L'objectif est de trouver une valeur cible pour le grand nombre, de sorte que le jeu s'arrête.

Listing 7.4 : *break* et *continue*

```
 1:  // Listing 7.4 - Instructions break et continue
 2:  #include <iostream>
 3:
 4:  int main()
 5:  {
 6:     using namespace std;
 7:
 8:     unsigned short petit;
 9:     unsigned long  grand;
10:     unsigned long  saut;
11:     unsigned long  cible;
12:     const unsigned short MAXPETIT = 65535;
13:
14:     cout << "Entrez un petit nombre : ";
15:     cin >> petit;
16:     cout << "Entrez un grand nombre : ";
17:     cin >> grand;
18:     cout << "Entrez un nombre pour les sauts : ";
19:     cin >> saut;
20:     cout << "Entrez un nombre cible : ";
21:     cin >> cible;
22:
23:     cout << "\n";
24:
25:     // configurer deux conditions d'arrêt pour la boucle
```

```
26:     while (petit < grand && petit < MAXPETIT)
27:     {
28:        petit++;
29:
30:        if (petit % saut == 0)  // sauter la décrémentation ?
31:        {
32:           cout << "Saut à " << petit << endl;
33:           continue;
34:        }
35:
36:        if (grand == cible)    // égalité avec la cible ?
37:        {
38:           cout << "Cible atteinte !";
39:           break;
40:        }
41:
42:        grand -= 2;
43:     }                   // fin de la boucle while
44:
45:     cout << "\nPetit nombre : " << petit <<
45a:            ", grand nombre : " << grand << endl;
46:     return 0;
47:  }
```

Ce programme produit le résultat suivant :

```
Entrez un petit nombre : 2
Entrez un grand nombre : 20
Entrez un nombre pour les sauts : 4
Entrez un nombre cible : 6

Saut à 4
Saut à 8

Petit nombre : 10, grand nombre : 8
```

Dans cet exemple, l'utilisateur a perdu, car `petit` a dépassé la valeur de `grand` avant que la valeur cible (6) ne soit atteinte.

La ligne 26 teste les conditions de l'itération. Si `petit` est inférieur à `grand` mais qu'il n'est pas supérieur à la valeur maximale autorisée, le corps de la boucle s'exécute.

La ligne 30 calcule le reste de la division de la valeur de `petit` par `saut`. Si `petit` est un multiple de `saut`, l'instruction `continue` s'exécute et le traitement revient donc au début de la boucle, en sautant le test de la cible et la décrémentation de `grand`.

La ligne 36 compare les valeurs de `cible` et de `grand` ; si elles sont égales, l'utilisateur a gagné. Le programme affiche alors un message et l'instruction `break` est atteinte et exécutée. Elle provoque une sortie immédiate de la boucle et l'exécution se poursuit à la ligne 44.



Les instructions `continue` et `break` doivent être utilisées avec précaution pour les mêmes raisons que l'instruction `goto`. Les programmes qui changent brutalement de direction sont plus difficiles à relire ; l'utilisation exagérée de `continue` et `break` peut rendre illisible une boucle `while`, même simple.
Le besoin de placer une instruction `break` dans une boucle indique souvent que la condition de fin de la boucle n'a pas été définie avec l'expression booléenne qui convient. Il vaut souvent mieux utiliser une instruction `if` pour sauter certaines lignes plutôt qu'une instruction `break`.

L'instruction `continue`

L'instruction `continue;` oblige le programme à revenir au début d'une boucle `while`, `do…while` ou `for`.

Le Listing 7.4 donne un exemple de son utilisation.

L'instruction `break`

`break;` provoque la sortie immédiate d'une boucle `while`, `do…while` ou `for`.

Exemple

```
while (condition)
{
    if (condition2)
        break;
    // instructions;
}
```

Les boucles *while* (*true*)

La condition testée au début d'une boucle `while` peut correspondre à n'importe quelle expression C++ valide. Tant qu'elle est vraie, l'itération se poursuit. Vous pouvez donc créer une boucle sans fin en utilisant la valeur booléenne `true` comme condition. Dans le Listing 7.5, nous utilisons cette fonctionnalité pour incrémenter un compteur jusqu'à 10.

Listing 7.5 : Exemple de boucle *while (true)*

```
 1:  // Listing 7.5
 2:  // Démonstration d'une boucle while (true)
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     int compteur = 0;
 8:
 9:     while (true)
10:     {
11:        compteur++;
12:        if (compteur > 10)
13:           break;
14:     }
15:     std::cout << "Compteur : " << compteur << std::endl;
16:     return 0;
17:  }
```

Ce programme produit le résultat suivant :

```
Compteur : 11
```

La ligne 9 configure la boucle `while` avec une condition qui ne pourra jamais être fausse. La variable compteur est incrémentée à la ligne 11, puis le programme vérifie si sa valeur est supérieure à 10. Dans le cas contraire, la boucle s'exécute à nouveau. En revanche, si le compteur est supérieur à 10, l'instruction `break` interrompt la boucle et va directement à la ligne 15. Le résultat est alors affiché.

Bien qu'il fonctionne correctement, ce programme est un bon exemple du mauvais choix d'un outil pour effectuer une tâche. En effet, il est préférable, dans ce cas, de tester la valeur du compteur dans la condition `while`.



Les boucles sans fin de type `while(true)` risquent de figer votre ordinateur si la condition de sortie n'est jamais atteinte. Il est donc conseillé de les utiliser avec précaution et de les tester soigneusement.

Avec C++, vous pouvez réaliser une même tâche de plusieurs façons. La véritable difficulté consiste à choisir l'outil adapté.

Faire	*Ne pas faire*
● Utiliser une boucle `while` pour répéter une tâche tant que la condition est vraie. ● Utiliser les instructions `break` et `continue` avec précaution. ● Vérifier que l'itération peut prendre fin.	● Utiliser l'instruction `goto`. ● Oublier la différence entre `continue` et `break`. `continue` revient au début, `break` va à la fin.

Les boucles *do...while*

Le corps d'une boucle `while` peut ne jamais s'exécuter car la condition est vérifiée *avant* tout traitement : si son résultat est égal à `false`, la séquence d'instructions ne sera donc pas traitée, comme le montre le Listing 7.6.

Listing 7.6 : Cas où le traitement de la boucle *while* ne s'exécute jamais

```
 1:  // Listing 7.6
 2:  // Exemple de saut du corps de la boucle while
 3:  // si la condition est false.
 4:
 5:  #include <iostream>
 6:
 7:  int main()
 8:  {
 9:     int compteur;
10:     std::cout << "Combien de bips voulez-vous afficher ? ";
11:     std::cin >> compteur;
12:     while (compteur > 0)
13:     {
14:        std::cout << "Bip !\n";
15:        compteur--;
16:     }
17:     std::cout << "Compteur à la sortie de la boucle : "
17a:               << compteur;
18:     return 0;
19:  }
```

Ce programme produit le résultat suivant :

```
Combien de bips voulez-vous afficher ?  2
Bip !
Bip !
Compteur à la sortie de la boucle : 0

Combien de bips voulez-vous afficher ?  0
Compteur à la sortie de la boucle : 0
```

L'utilisateur entre une valeur de départ (ligne 10) qui est copiée dans la variable entière `compteur` dont la valeur est testée à la ligne 12, puis décrémentée dans le corps de la boucle `while`. Dans le résultat, vous pouvez constater qu'à la première exécution du programme, l'utilisateur a entré 2 et que le corps de la boucle s'est exécuté deux fois. Dans le second exemple, l'utilisateur a tapé 0. La valeur du compteur a été testée à la ligne 12. La condition était fausse puisque `compteur` n'était pas supérieur à 0. La séquence d'instructions ne s'est donc pas exécutée et aucun `bip` ne s'est affiché à l'écran.

Comment faire pour afficher au moins un bip ? Vous pouvez placer une instruction `if` avant la boucle `while` afin d'affecter une valeur minimale à `compteur` afin d'entrer dans la boucle :

```
if (compteur < 1)  // affecte une valeur minimale
    compteur = 1;
```

Mais cette solution est un peu boiteuse et inélégante.

L'instruction *do...while*

Une boucle `do...while` garantit que les instructions de la boucle s'exécuteront *au moins une* fois, car le corps de la boucle est exécuté *avant* le test de la condition. Le code du Listing 7.7 correspond au programme du listing précédent, mais utilise une boucle `do...while`.

Listing 7.7 : Exemple de boucle *do...while*

```
 1:  // Listing 7.7
 2:  // Boucle do while
 3:
 4:  #include <iostream>
 5:
 6:  int main()
 7:  {
 8:     using namespace std;
 9:     int compteur;
10:     cout << "Combien de bips voulez-vous afficher ? ";
11:     cin >> compteur;
12:     do
13:     {
14:        cout << "Bip !\n";
15:        compteur--;
16:     }  while (compteur >0 );
17:     cout << "Compteur à la sortie de la boucle : "
17a:         << counter << endl;
18:     return 0;
19:  }
```

Ce programme produit le résultat suivant :

```
Combien de bips voulez-vous afficher ?  0
Bip !
Compteur à la sortie de la boucle : -1
```

Comme le programme précédent, le Listing 7.7 affiche le mot "Bip" sur la console le nombre de fois indiqué. Par contre, il l'affichera toujours au moins une fois.

La ligne 10 demande à l'utilisateur d'entrer une valeur de départ qui est stockée dans la variable entière `compteur`. Le corps de la boucle étant exécuté avant le test de la condition, un bip est affiché à la ligne 14. La ligne suivante décrémente le compteur et la ligne 16 teste la condition. Si elle est vérifiée, le programme retourne au début de la boucle, à la ligne 14. Dans le cas contraire, c'est la ligne 17 qui s'exécute.

Les instructions `continue` et `break` remplissent le même rôle que dans une boucle `while`. La seule différence entre une boucle `while` et une boucle `do…while` concerne le moment où la condition est testée.

L'instruction `do...while`

La syntaxe de l'instruction `do...while` est la suivante :

```
do
  instruction
while (condition);
```

L'instruction est exécutée, *puis* la condition est évaluée. Si cette dernière renvoie `true`, le programme effectue un autre passage dans la boucle, sinon le traitement prend fin. Une instruction `do...while` peut inclure les mêmes instructions et conditions qu'une boucle `while`.

Exemple 1

```
// incrémentation jusqu'à 10
int x = 0;
do
    cout << "X : " << x++;
while (x < 10);
```

Exemple 2

```
// affichage de l'alphabet en minuscules.
char lettre = 'a';
do
{
    cout << lettre << ' ';
    lettre++;
} while ( lettre <= 'z' );
```

Faire	*Ne pas faire*
● Utiliser `do...while` lorsque la boucle doit s'exécuter au moins une fois. ● Utiliser une boucle `while` pour sauter le corps de la boucle si la condition est fausse. ● Vérifier toutes les boucles pour vous assurer qu'elles effectuent le traitement attendu.	● Utiliser `break` et `continue` avec des boucles, à moins que l'action du code ne soit évidente. Il y a souvent des méthodes plus claires pour accomplir ces opérations. ● Utiliser l'instruction `goto`.

Les boucles *for*

Dans une boucle `while`, il est souvent nécessaire de réaliser trois étapes : définir une condition de départ, déterminer si elle est vraie et incrémenter une variable ou y affecter une valeur différente à chaque passage. C'est ce que l'on fait dans le Listing 7.8.

Listing 7.8 : Boucle *while* revisitée

```
 1:  // Listing 7.8
 2:  // Traitement itératif avec while
 3:
 4:  #include <iostream>
 5:
 6:  int main()
 7:  {
 8:     int compteur = 0;
 9:
10:     while (compteur < 5)
11:     {
12:        compteur++;
13:        std::cout << "Bip !  ";
14:     }
15:
16:     std::cout << "\nCompteur à la sortie de la boucle : "
16a:               << compteur << std::endl;
17:     return 0;
18:  }
```

Ce programme produit le résultat suivant :

```
Bip !  Bip !  Bip !  Bip !  Bip !
Compteur à la sortie de la boucle : 5.
```

Dans ce listing, on voit bien les trois étapes mentionnées plus haut. La condition de départ est définie à la ligne 8 : la variable `compteur` est initialisée à 0. Deux lignes plus loin, le programme détermine si elle est inférieure à 5. Enfin, la variable `compteur` est incrémentée à la ligne 12. Cette boucle affiche un simple message à la ligne 13, mais ce traitement pourrait être plus compliqué.

Une boucle `for` combine ces trois opérations (initialisation, test et incrémentation) en une seule instruction. La syntaxe est simple : il suffit d'utiliser le mot-clé `for` suivi de parenthèses `()` entre lesquelles figurent les trois opérations séparées les unes des autres par un point-virgule (`;`).

```
for ( initialisation ; test ; action )
{
   ...
}
```

La première expression, *initialisation*, est la condition de départ, ou initialisation ; elle peut contenir n'importe quelle instruction C++ valide, mais l'usage veut que l'on y crée et initialise une variable compteur. La deuxième expression, *test*, correspond au test, mais rien ne vous empêche d'y placer une instruction C++ différente. Elle remplit le même rôle que l'expression conditionnelle d'une boucle `while`. Enfin, la troisième expression, *action*, correspond à l'action à réaliser. Cette action concerne généralement l'augmentation ou la baisse d'une valeur. Là aussi, vous pouvez indiquer une instruction permettant d'effectuer une autre tâche. Le Listing 7.9 utilise une boucle `for` pour effectuer le même traitement que le Listing 7.8.

Listing 7.9 : Exemple de boucle *for*

```
 1:  // Listing 7.9
 2:  // Traitement itératif avec for
 3:
 4:  #include <iostream>
 5:
 6:  int main()
 7:  {
 8:     int compteur;
 9:     for (compteur = 0; compteur < 5; compteur++)
10:        std::cout << "Bip ! ";
11:
12:     std::cout << "\nCompteur à la sortie de la boucle : "
12a:              << compteur << std::endl;
13:     return 0;
14:  }
```

Ce programme produit le résultat suivant :

```
Bip !  Bip !  Bip !  Bip !  Bip !
Compteur à la sortie de la boucle : 5.
```

L'instruction `for` rassemble trois opérations à la ligne 9 : le compteur est initialisé à zéro, le test porte sur la valeur de cette variable qui doit être inférieure à 5 et enfin, le compteur est incrémenté de un point à chaque passage. Le corps de la boucle consiste en une seule instruction à la ligne 10, mais, il est possible de placer un bloc d'instructions.

L'instruction `for`

La syntaxe de l'instruction `for` est la suivante :

```
for (initialisation; test; action)
    instruction;
```

L'instruction *initialisation* permet d'affecter une valeur au compteur ou de préparer le traitement itératif. Le *test* (n'importe quelle expression C++) est évalué à chaque passage dans la boucle. Si le résultat est vrai, le corps de la boucle s'exécute puis l'instruction spécifiée dans *action* (en général, le compteur est incrémenté).

Exemple 1

```
// afficher Bonjour 10 fois
for (int i = 0; i<10; i++)
    cout << "Bonjour ! ";
```

Exemple 2

```
for (int i = 0; i < 10; i++)
{
    cout << "Bonjour !" << endl;
    cout << "i est égal a " << i << endl;
}
```

Boucles *for* évoluées

L'instruction `for` est très puissante et d'une utilisation très souple. Ses trois éléments (*initialisation*, *test* et *action*) autorisent un grand nombre de variations.

Initialisations et incrémentations multiples

Il n'est pas rare d'initialiser plusieurs variables, de tester une expression logique complexe et d'exécuter plusieurs instructions. Vous avez la possibilité de remplacer l'initialisation et

l'action par plusieurs instructions C++, séparées les unes des autres par une virgule, comme dans le Listing 7.10

Listing 7.10 : Boucle *for* contenant plusieurs instructions

```
 1:  //Listing 7.10
 2:  // Boucle for comportant plusieurs
 3:  // instructions
 4:  #include <iostream>
 5:
 6:  int main()
 7:  {
 8:
 9:   for (int i=0, j=0; i<3; i++, j++)
10:     std::cout << "i : " << i << ", j : " << j << std::endl;
11:   return 0;
12:  }
```

Ce programme produit le résultat suivant :

```
i : 0 , j : 0
i : 1 , j : 1
i : 2 , j : 2
```

La ligne 9 initialise à zéro les deux variables `i` et `j` en séparant par une virgule les deux expressions. Vous pouvez également constater que ces initialisations sont séparées de la condition de test par le point-virgule attendu.

Lorsque ce programme s'exécute, le test (`i<3`) est évalué. Le résultat étant vrai, le corps de l'instruction `for` s'exécute et les valeurs apparaissent à l'écran. Puis, la troisième clause de l'instruction `for` est exécutée. Cette instruction comporte également deux expressions pour incrémenter `i` et `j`.

Après exécution de l'instruction de la ligne 10, la condition est de nouveau évaluée. Si elle est toujours vraie, les actions se répètent (`i` et `j` sont à nouveau incrémentées), puis la boucle s'exécute une nouvelle fois. Lorsque la condition est fausse, l'action n'est pas exécutée et le programme quitte la boucle.

Utilisation d'instructions nulles dans les boucles *for*

Toute instruction d'une boucle `for` peut être omise. Pour cela, utilisez une instruction nulle en insérant simplement un point-virgule à l'endroit où aurait figuré la clause. Dans le Listing 7.11, la première et la troisième clause sont nulles : la boucle `for` agit alors comme une boucle `while`.

Listing 7.11 : Instructions nulles dans une boucle *for*

```
 1:  // Listing 7.11
 2:  // Boucle For avec des instructions nulles
 3:
 4:  #include <iostream>
 5:
 6:  int main()
 7:  {
 8:     int compteur = 0;
 9:
10:     for( ; compteur < 5; )
11:     {
12:        compteur++;
13:        std::cout << "Bip !  ";
14:     }
15:
16:     std::cout << "\nCompteur à la sortie de la boucle : "
16a:               << compteur << std::endl;
17:     return 0;
18:  }
```

Ce programme produit le résultat suivant :

```
Bip !  Bip !  Bip !  Bip !  Bip !
Compteur à la sortie de la boucle : 5.
```

Vous constatez que cette boucle fonctionne exactement comme celle du Listing 7.8. La ligne 8 initialise la variable `compteur`. À la ligne 10, l'instruction `for` n'initialise pas de valeur, mais vérifie simplement si le compteur est inférieur à 5. L'incrémentation se fait dans le corps de la boucle, ce qui revient à écrire :

```
while (compteur < 5)
```

Cette fois encore, vous pouvez constater que C++ permet de réaliser une même tâche de plusieurs façons. Un programmeur expérimenté n'utiliserait pas une instruction `for` comme celle du Listing 7.11 : cet exemple n'a qu'une valeur de démonstration de la souplesse de l'instruction `for`. En fait, comme le montre le Listing 7.12, il est possible d'utiliser des instructions `break` et `continue` pour créer une boucle `for` vide.

Listing 7.12 : Instruction *for* vide

```
1:  //Listing 7.12 : illustration
2:  //d'une boucle for vide
3:
```

```
 4:  #include <iostream>
 5:
 6:  int main()
 7:  {
 8:     int compteur = 0;           // initialisation
 9:     int max;
10:     std::cout << "Combien de bips voulez-vous afficher ? ";
11:     std::cin >> max;
12:     for (;;)                 // boucle for sans fin
13:     {
14:        if (compteur < max)    // test
15:        {
16:           std::cout << "Bip ! " << std::endl;
17:           compteur++;         // incrémentation
18:        }
19:        else
20:           break;
21:     }
22:     return 0;
23:  }
```

Ce programme produit le résultat suivant :

```
Combien de bips voulez-vous afficher ? 3
Bip !
Bip !
Bip !
```

Cette boucle `for` est dépouillée à l'extrême car l'instruction ne contient aucune des trois clauses habituelles *`initialisation`*, *`test`* et *`action`*. Tout d'abord, l'initialisation s'effectue à la ligne 8 avant que la boucle `for` ne commence. Le test est réalisé dans une instruction distincte, à la ligne 14. Si le résultat est vrai, l'incrémentation s'effectue à la ligne 17. Dans le cas contraire, le programme quitte la boucle pour aller directement à la ligne 20.

Bien entendu, ce programme est quelque peu absurde mais, dans certains cas, une boucle `for(;;)` ou une boucle `while (true)` répondront parfaitement à vos besoins. Avec l'instruction `switch`, vous allez apprendre à utiliser correctement ces types de boucles.

Boucles *for* vides

Parfois, l'instruction `for` se suffit à elle-même pour réaliser un traitement. Le corps de la boucle devient alors inutile et, dans ce cas, vous devez insérer une instruction nulle (`;`). Le point-virgule peut figurer sur la même ligne que l'instruction `for`, mais il est alors facile de l'oublier. Le Listing 7.13 présente un exemple de boucle `for` vide.

Listing 7.13 : Corps d'une boucle *for* vide

```
 1:  // Listing 7.13
 2:  // Exemple de corps de la boucle for
 3:  // composé d'une instruction nulle
 4:
 5:  #include <iostream>
 6:  int main()
 7:  {
 8:   for (int i=0; i<5; std::cout << "I : " << i++ << std::endl)
 9:        ;         // instruction vide (corps de la boucle)
10:    return 0;
11:  }
```

Ce programme produit le résultat suivant :

```
i : 0
i : 1
i : 2
i : 3
i : 4
```

L'instruction `for` de la ligne 8 comprend trois éléments : l'instruction d'initialisation définit le compteur `i`, puis l'initialise à zéro. L'instruction de test vérifie si `i` est inférieure à 5. Enfin, l'instruction d'action affiche la valeur de `i` et incrémente le compteur.

Comme il n'y a plus rien à faire dans le corps de la boucle, on utilise l'instruction nulle (`;`). Notez que cette boucle n'est pas très bien construite, car l'instruction d'action peut poser des problèmes de compréhension au moment de la compilation. Il serait préférable de taper les instructions comme suit :

```
8:         for (int i = 0; i<5; i++)
9:              cout << "i : " << i << endl;
```

Le résultat est le même, mais avouez que c'est plus clair.

Imbrication de boucles

N'importe quelle boucle peut être imbriquée dans le corps d'une autre. Une boucle interne s'exécute en totalité à chaque itération de la boucle de niveau immédiatement supérieur. Le Listing 7.14 utilise deux boucles `for` imbriquées pour afficher une matrice.

Listing 7.14 : Boucles *for* imbriquées

```
 1:  //Listing 7.14
 2:  //Boucles imbriquées
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     using namespace std;
 8:     int lignes, colonnes;
 9:     char car;
10:     cout << "Nombre de lignes ? ";
11:     cin >> lignes;
12:     cout << "Nombre de colonnes ? ";
13:     cin >> colonnes;
14:     cout << "Caractère à utiliser ? ";
15:     cin >> car;
16:     for (int i = 0; i < lignes; i++)
17:     {
18:        for (int j = 0; j < colonnes; j++)
19:           cout << car ;
20:        cout << endl;
21:     }
22:     return 0;
23:  }
```

Ce programme produit le résultat suivant :

```
Nombre de lignes : 4
Nombre de colonnes : 12
Caractère à utiliser ? X
XXXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXX
```

L'utilisateur est invité à entrer le nombre de lignes et le nombre de colonnes, puis le caractère qui va se répéter dans toutes les cases du tableau. La première boucle `for`, à la ligne 16, initialise un compteur (`i`) à 0, puis exécute le corps de la boucle `for` externe.

La ligne 18 est la première ligne du corps de la boucle `for` externe, elle met en place une autre boucle `for`. Cette boucle imbriquée initialise à zéro un deuxième compteur (`j`) et exécute son corps. La ligne 19 affiche le caractère choisi et l'exécution revient à l'en-tête de la boucle imbriquée. Vous remarquerez que la boucle imbriquée n'a qu'une seule instruction (l'affichage du caractère). Sa condition (`j < colonnes`) est évaluée ; si elle est

vraie, `j` est incrémentée et le caractère suivant est affiché. Ce traitement s'arrête lorsque `j` est égale au nombre de colonnes souhaité.

Lorsque le test de la boucle imbriquée renvoie un résultat faux (ici, après l'affichage de 12 `X`), le programme saute à la ligne 20 et affiche un retour à la ligne. La boucle externe revient alors à son en-tête pour tester la condition `i < lignes`. Si le résultat est vrai (`i` est inférieur au nombre de lignes), `i` est incrémentée et le corps de la boucle s'exécute.

À la deuxième itération de la boucle externe, la boucle imbriquée s'exécute à nouveau ; `j` est réinitialisée à 0 et le traitement complet de la boucle recommence.

Le point important à noter, ici, est qu'une boucle imbriquée s'exécute complètement à chaque itération de la boucle extérieure. Dans notre exemple, le caractère `X` s'affiche donc `colonnes` fois pour chaque ligne.



Nombreux sont les programmeurs qui utilisent les lettres `i` *et* `j` *comme variables de compteur. Cette habitude provient du FORTRAN, qui n'autorisait que les lettres* `i`*,* `j`*,* `k`*,* `l`*,* `m` *et* `n` *comme noms de variables de comptage.*
Même si cela peut sembler inoffensif, vos lecteurs pourraient s'interroger sur l'objectif du compteur et l'utiliser à mauvais escient. Un programme complexe contenant des boucles imbriquées devient encore plus confus. En ce cas, il vaut mieux préciser le but de la variable compteur dans son nom, en utilisant par exemple `IndiceClient` *ou* `CompteurEntrees`*.*

Portée des variables dans une boucle *for*

Par le passé, la portée des variables déclarées dans une boucle `for` s'étendait au bloc extérieur. La norme ANSI a modifié ce comportement en réduisant la portée de ces variables au bloc de la boucle `for` ; cependant, certains compilateurs ne respectent pas encore cette spécification. Le mieux est de tester le vôtre avec le code suivant :

```
#include <iostream>
int main()
{
     // portée de i limitée à la boucle for ?
     for (int i = 0; i < 5; i++)
     {
         std::cout << "i : " << i << std::endl;
     }

     i = 7;  // ne devrait pas être accessible !
     return 0;
}
```

Si la compilation se déroule sans problème, votre compilateur ne prend pas en compte ce nouvel aspect de la norme ANSI.

Si votre compilateur signale que `i` n'est pas encore défini (à la ligne `i = 7`), il tient compte de ce nouveau standard. Le code suivant sera compilé dans les deux cas si vous déclarez `i` en dehors de la boucle, comme ici :

```
#include <iostream>
int main()
{
     int i; // i déclaré hors de la boucle for
     for (i = 0; i < 5; i++)
     {
         std::cout << "i : " << i << std::endl;
     }

     i = 7;  // i est maintenant accessible dans tous les cas
     return 0;
}
```

Résumé des boucles

Au Chapitre 5, nous avons résolu la suite de Fibonacci en faisant appel à la récursivité. Rappelons que cette suite commence par les chiffres 1, 1, 2, 3, les autres valeurs étant la somme des deux nombres précédents. Exemple :

```
1, 1, 2, 3, 5, 8, 13, 21, 34...
```

Le nombre de Fibonacci *n* est la somme du nombre *n*-1 et du nombre *n*-2. Le Listing 7.15 présente une solution à ce problème en utilisant une itération.

Listing 7.15 : Déterminer la valeur d'un nombre de Fibonacci à l'aide d'une itération

```
 1:  // Listing 7.15 - Déterminer la valeur du nième nombre
 2:  // à l'aide d'une itération
 3:
 4:  #include <iostream>
 5:
 6:  unsigned int fib(unsigned int position);
 7:  int main()
 8:  {
 9:     using namespace std;
10:     unsigned int reponse, position;
11:     cout << "Quelle position ? ";
12:     cin >> position;
```

```
13:     cout << endl;
14:
15:     reponse = fib(position);
16:     cout << reponse << " est le ";
17:     cout << position << "e nombre dans la suite. " << endl;
18:     return 0;
19: }
20:
21: unsigned int fib(unsigned int n)
22: {
23:     unsigned int moinsDeux=1, moinsUn=1, reponse=2;
24:
25:     if (n < 3)
26:        return 1;
27:
28:     for (n -= 3; n != 0; n--)
29:     {
30:        moinsDeux = moinsUn;
31:        moinsUn = reponse;
32:        reponse = moinsUn + moinsDeux;
33:     }
34:
35:     return reponse;
36: }
```

Ce programme produit le résultat suivant :

```
Quelle position ? 4
3 est le 4e nombre dans la suite.

Quelle position ?  5
5 est le 5e nombre dans la suite.

Quelle position ? 20
6765 est le 20e nombre dans la suite.
```

Dans le Listing 7.15, nous avons préféré l'itération à la récursivité, car cette approche est plus rapide et utilise moins de mémoire que la solution récursive.

La ligne 11 demande à l'utilisateur de saisir la position du nombre à trouver. Puis, on appelle la fonction `fib()` pour calculer la valeur correspondante. Si la position est inférieure à 3, la fonction renvoie 1. À partir de la position 3, la fonction utilise l'algorithme suivant :

1. Déterminer la position de départ : affecter la valeur 2 à la variable `reponse`, la valeur 1 à `moinsDeux` et la valeur 1 à `moinsUn`. On ôte 3 à la position, car les deux premiers nombres sont pris en charge par la position de départ.

2. Pour chaque nombre, déterminer la valeur dans la suite. Pour cela :
   a. Transférer la valeur de `moinsUn` dans `moinsDeux`.
   b. Copier la valeur de `reponse` dans `moinsUn`.
   c. Faire la somme de `moinsUn` et de `moinsDeux`, puis copier le résultat dans la variable `reponse`.
   d. Décrémenter `n`.
3. Lorsque `n` est égal à 0, renvoyer le résultat.

C'est comme si vous résolviez le problème avec du papier et un crayon. Pour le cinquième nombre de Fibonacci, vous écririez d'abord :

```
1, 1, 2,
```

puis vous vous diriez qu'il y a encore deux nombres de plus à trouver. Vous additionneriez alors 2 et 1 et écririez 3 : il ne resterait alors plus qu'un nombre à trouver. Pour cela, vous additionneriez 3 et 2 et vous écririez 5. En pratique, vous reportez à chaque fois votre attention d'un nombre vers la droite et vous décrémentez le nombre de valeurs restant à trouver.

Remarquez la condition qui est testée à la ligne 28 (`n != 0`). De nombreux programmeurs utiliseraient plutôt cette notation :

```
for ( n-=3; n; n-- )
```

Au lieu d'utiliser une condition relationnelle, seule la valeur de `n` sert à la condition dans l'instruction `for`. Il s'agit d'une expression caractéristique de C++ ; `n` est considéré comme étant équivalent à l'expression `n != 0`. Se contenter d'utiliser `n` s'appuie sur le fait que lorsque `n` est égal à 0, cette valeur est considérée comme fausse par C++. Pour se conformer à la norme actuelle de C++, il vaut mieux utiliser une véritable condition pour évaluer la valeur `false` plutôt qu'utiliser une valeur numérique.

Compilez, exécutez le programme et comparez le avec la solution récursive du Chapitre 5. Tapez 25 comme position dans la suite de Fibonacci et comparez les temps d'exécution des deux versions : vous pouvez constater que le programme itératif tourne bien plus rapidement que le programme récursif car ce dernier réalise de nombreux appels de fonctions, qui ont un impact sur les performances. En outre, la solution récursive, bien qu'élégante, effectue plusieurs fois les mêmes calculs. Les ordinateurs modernes optimisant les traitements arithmétiques, la solution itérative devrait être très rapide.

Attention à la taille du nombre saisi car le nombre de Fibonacci augmente rapidement : même la capacité des entiers longs non signés sera vite dépassée.

Contrôler le flux avec des instructions *switch*

Au Chapitre 4, nous avons étudié les instructions `if` et `if...else`. Lorsque leur niveau d'imbrication devient trop élevé, il est préférable d'utiliser l'instruction `switch`. À la différence de `if`, qui ne permet de tester qu'une seule condition, une instruction `switch` permet d'effectuer un "aiguillage" dans le programme en fonction des différentes valeurs d'une expression. Cette instruction se présente sous la forme suivante :

```
switch (expression)
{
case valeur1: instruction;
                    break;
case valeur2: instruction;
                    break;
....
case valeurN:   instruction;
                    break;
default:       instruction;
}
```

expression correspond à n'importe quelle expression C++ renvoyant une valeur entière. Les *instructions* sont des instructions ou des blocs d'instructions C++. Les différentes valeurs des clauses `case` sont des valeurs entières (ou des expressions pouvant être sans ambiguïté convertie en une valeur entière). L'instruction `switch` évalue la valeur d'*expression* puis compare le résultat avec les valeurs des clauses `case`. Cette comparaison s'effectue toujours selon l'égalité, les opérateurs de comparaison et booléens sont interdits.

Si l'une des valeurs de `case` correspond à l'expression, le programme saute vers les instructions correspondantes, puis continue l'exécution jusqu'à la fin de l'instruction `switch`, à moins qu'il ne rencontre une instruction `break`. Si aucune correspondance n'est trouvée, il va directement à l'instruction `default`, qui est facultative ; s'il n'y a ni valeur `case` correspondante, ni de clause `default`, le programme quitte le bloc d'instructions `switch`.



Il est toujours judicieux d'utiliser une instruction `default` dans un bloc d'instructions `switch`. Même si vous n'avez pas de valeur par défaut à traiter, cette instruction pourra servir à tester un cas qui survient rarement ou à afficher un message d'erreur. Elle peut également être très pratique lors de la mise au point de l'application.

Il est important de noter que si une clause `case` ne se termine pas par `break`, l'exécution du programme se poursuit avec la clause `case` suivante. C'est parfois nécessaire, mais il s'agit généralement d'un oubli du programmeur, qui provoquera un comportement inattendu.

Si vous décidez sciemment d'autoriser ce passage à la clause `case` suivante, n'oubliez pas de placer un commentaire pour indiquer qu'il ne s'agit pas d'un oubli.

Le Listing 7.16 illustre l'utilisation de l'instruction `switch`.

Listing 7.16 : Utilisation de l'instruction *switch*

```
 1:  //Listing 7.16
 2:  // L'instruction switch
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     using namespace std;
 8:     unsigned short int nombre;
 9:     cout << "Entrez un nombre entre 1 et 5 : ";
10:     cin >> nombre;
11:     switch (nombre)
12:     {
13:     case 0:  cout << "Trop petit, à refaire !";
14:              break;
15:     case 5:  cout << "Bravo ! " << endl;      // case suivant
16:     case 4:  cout << "Bon choix ! " << endl;  // case suivant
17:     case 3:  cout << "Excellent ! " << endl;  // case suivant
18:     case 2:  cout << "Formidable ! " << endl; // case suivant
19:     case 1:  cout << "Incroyable ! " << endl;
20:              break;
21:     default: cout << "Trop grand !" << endl;
22:              break;
23:     }
24:     cout << endl << endl;
25:     return 0;
26:  }
```

Ce programme produit le résultat suivant :

```
Entrez un nombre entre 1 et 5 :  3
Excellent !
Formidable !
Incroyable !

Entrez un chiffre entre 1 et 5 : 8
Trop grand !
```

Aux lignes 9 et 10, l'utilisateur est invité à entrer un nombre qui sera ensuite transmis à l'instruction `switch` en ligne 11. Si la valeur saisie est zéro, le programme saute à l'instruction `case` de la ligne 13. Le message "Trop petit, à refaire !" apparaît alors et l'instruction `break` de la ligne 14 met fin au `switch`. Si la valeur est égale à 5, l'exécution passe à la ligne 15 et les instructions `case` qui suivent s'exécutent les unes après les autres jusqu'à l'instruction `break` de la ligne 20, qui met fin au `switch`.

Le nombre de messages affichés est donc égal à la valeur saisie. Si cette dernière est supérieure à 5, la clause `default` s'exécute et le message "Trop grand !" apparaît (ligne 21).

L'instruction `switch`

La syntaxe de l'instruction `switch` est la suivante :

```
switch (expression)
{
  case valeur1: instruction;
  case valeur2: instruction;
....
  case valeurN: instruction
  default: instruction;
}
```

L'instruction `switch` permet d'aiguiller le programme en différent points en fonction de la valeur de *expression*. Après l'évaluation de l'expression, le pointeur d'instruction va directement à clause `case` dont la valeur correspond. L'exécution du programme se poursuit jusqu'à la fin de l'instruction `switch` ou jusqu'à ce qu'une instruction `break` soit rencontrée.

Si l'expression ne correspond à aucune clause `case`, et qu'il y a une clause `default`, le programme exécute cette clause ; sinon, l'instruction `switch` se termine.

Exemple 1

```
switch (choix)
{
    case 0:
        cout << "Zéro !" << endl;
        break
    case 1:
        cout << "Un !" << endl;
        break;
    case 2:
        cout << "Deux !" << endl;
    default:
        cout << "Clause par défaut !" << endl;
}
```

Exemple 2

```
switch (choix)
{
   case 0:
   case 1:
   case 2:
       cout << "Inférieur à 3 !";
       break;
   case 3:
       cout << "Égal à 3 !";
       break;
   default:
       cout << "Supérieur à 3 !";
}
```

Utilisation de *switch* dans un menu

Le Listing 7.17 reprend le principe de la boucle `for(;;)` décrite plus haut. Une boucle qui ne se termine pas est appelée *boucle sans fin*, car elle se répétera continuellement si elle ne rencontre pas une instruction `break`. Dans le Listing 7.17, le programme affiche un menu, invite l'utilisateur à faire un choix, agit en fonction de ce choix, puis affiche de nouveau le menu. Pour mettre fin à ce traitement, l'utilisateur doit choisir l'option Quitter.



Certains programmeurs aiment écrire :

```
#define EVER ;;
for (EVER)
{
   // instructions…
}
```

Une boucle *sans fin* ne contient pas de condition de sortie. Seule une instruction `break` permet d'en sortir. Les boucles sans fin sont également appelées boucles infinies.

Listing 7.17 : Exemple de boucle sans fin

```
 1:  //Listing 7.17
 2:  //Boucle sans fin permettant une interaction
2a:  // avec l'utilisateur à l'aide d'un menu
 3:  #include <iostream>
 4:
```

```
 5:  // prototypes
 6:  int menu();
 7:  void Tache();
 8:  void AutreTache(int);
 9:
10:  using namespace std;
11:
12:  int main()
13:  {
14:     bool fin = false;
15:     for (;;)          // début de la boucle sans fin
16:     {
17:        int choix = menu();
18:        switch(choix)
19:        {
20:        case (1):
21:           Tache();
22:           break;
23:        case (2):
24:           AutreTache(2);
25:           break;
26:        case (3):
27:           AutreTache(3);
28:           break;
29:        case (4):
30:           continue;         // redondant!
31:           break;
32:        case (5):
33:           fin = true;
34:           break;
35:        default:
36:           cout << "Choix non valide, recommencez ! " << endl;
37:           break;
38:        }                    // fin de switch
39:
40:        if (fin == true)
41:           break;
42:     }                       // fin de la boucle
43:     return 0;
44:  }                          // fin de main()
45:
46:  int menu()
47:  {
48:     int choix;
49:
50:     cout << " **** Menu ****  "  << endl << endl;
51:     cout << "(1) Choix un. "  << endl;
```

```
52:     cout << "(2) Choix deux. " << endl;
53:     cout << "(3) Choix trois. " << endl;
54:     cout << "(4) Réafficher le menu. " << endl;
55:     cout << "(5) Quitter. "  << endl << endl;
56:     cout << "Votre choix : ";
57:     cin >> choix;
58:     return choix;
59:  }
60:
61:  void Tache()
62:  {
63:     cout << "Tâche une ! " << endl;
64:  }
65:
66:  void AutreTache(int combien)
67:  {
68:     if (combien == 2)
69:        cout << "Tâche deux ! " << endl;
70:     else
71:        cout << "Tâche trois! " << endl;
72:  }
```

Ce programme produit le résultat suivant :

```
**** Menu ****
 (1) Choix un.
 (2) Choix deux.
 (3) Choix trois.
 (4) Réafficher le menu
 (5) Quitter.

Votre choix : 1
Tâche une !

**** Menu ****
 (1) Choix un.
 (2) Choix deux.
 (3) Choix trois.
 (4) Réafficher le menu
 (5) Quitter.

Votre choix : 3
Tâche trois !

**** Menu ****
 (1) Choix un.
 (2) Choix deux.
```

```
(3) Choix trois.
(4) Réafficher le menu
(5) Quitter.

Votre choix : 5
```

Ce programme rassemble des concepts abordés dans ce chapitre (comme l'instruction `switch`) et dans les chapitres précédents.

La boucle sans fin commence à la ligne 15. La fonction `menu()` affiche le menu à l'écran et récupère le choix de l'utilisateur. Cette valeur est traitée dans l'instruction `switch` qui s'étend de la ligne 18 à la ligne 38.

Si l'utilisateur choisit 1, le programme va directement à la clause `case (1):` de la ligne 20, puis il exécute la fonction `Tache()`, qui affiche un message. L'instruction `break` de la ligne 22 interrompt l'exécution de l'instruction `switch` et le programme va alors directement à la ligne 39. À la ligne suivante, la variable `fin` est évaluée. Si elle vaut `true`, l'instruction `break` de la ligne 41 s'exécute et la boucle `for(;;)` prend fin. Dans le cas contraire, l'exécution reprend au début de la boucle (ligne 15).

L'instruction `continue` de la ligne 30 est redondante. Si cette instruction n'était pas mentionnée, l'instruction `break` suivante mettrait fin à l'instruction `switch`, le résultat du test sur `fin` serait faux et un nouveau passage dans la boucle aurait lieu, ce qui réafficherait le menu. L'instruction `continue` permet toutefois d'éviter le test de `fin`.

Faire	*Ne pas faire*
● Documenter toutes les clause `case` ne contenant pas (sciemment) d'instruction `break`. ● Prévoir une clause `default` dans les instructions `switch`, afin de traiter les situations exceptionnelles.	● Utiliser des instructions `if…else` complexes alors qu'une instruction `switch` plus claire conviendrait. ● Oublier l'instruction `break` dans une suite de clauses `case`, sauf si vous voulez que le traitement se poursuive dans la clause `case` suivante.

Questions-réponses

Q Comment choisir entre l'instruction `if…else` et l'instruction `switch` ?

R Si la condition comprend plus d'une ou deux clauses `else` testant la même valeur, préférez l'instruction `switch`.

Q Comment choisir entre l'instruction `while` et l'instruction `do...while` ?

R Si le corps de l'itération doit s'exécuter au moins une fois, choisissez l'instruction `do...while`. Dans le cas contraire, préférez l'instruction `while`.

Q Comment choisir entre l'instruction `while` et l'instruction `for` ?

R L'instruction `for` permet d'initialiser une variable compteur, de tester cette variable, puis de l'incrémenter à chaque passage. En revanche, si la variable est déjà initialisée et n'a pas besoin d'être incrémentée à chaque itération, préférez l'instruction `while`. Les programmeurs expérimentés s'attendent à cette utilisation et auront plus de mal à comprendre votre code si vous n'agissez pas ainsi.

Q Doit-on préférer l'instruction `while (true)` à l'instruction `for(;;)` ?

R Non. Ces instructions permettent toutes les deux de définir une boucle sans fin, mais il vaut mieux les éviter.

Q Pourquoi ne faut-il pas utiliser une variable comme condition, par exemple `while(n)` ?

R Dans la norme C++ actuelle, une expression renvoie une valeur booléenne `true` ou `false`. Vous pouvez associer `false` à 0 et `true` à une autre valeur, mais il vaut mieux utiliser une expression qui renvoie l'une des deux valeurs booléennes.

Testez vos connaissances

1. Comment peut-on initialiser plusieurs variables dans une boucle `for` ?
2. Pourquoi faut-il éviter d'utiliser l'instruction `goto` ?
3. Est-il possible de créer une boucle `for` dont la séquence d'instructions ne s'exécute jamais ?
4. Lorsque cette boucle `for` se termine, quelle est la valeur de `x` ?

```
for (int x = 0; x < 100; x++)
```

5. Est-il possible d'imbriquer des boucles `while` dans des boucles for ?
6. Est-il possible de créer une boucle qui ne se termine jamais ? Donnez un exemple.
7. Que se passe-t-il si vous créez une boucle sans fin ?

Exercices

1. Écrivez une boucle `for` imbriquée qui produit une matrice de 10 fois 10 zéros.
2. Comptez de 100 à 200, de deux en deux en utilisant une boucle `for`.
3. Même exercice que ci-dessus mais avec une boucle `while`.
4. Même exercice que ci-dessus mais avec une boucle `do...while`.
5. CHERCHEZ L'ERREUR : pourquoi ce code est-il bogué ?

```
int compteur = 0
while (compteur < 10)
{
    cout << "Compteur : " << compteur;
}
```

6. CHERCHEZ L'ERREUR : pourquoi ce code est-il bogué ?

```
for (int compteur = 0; compteur < 10; compteur++);
    cout << compteur << "\n ";
```

7. CHERCHEZ L'ERREUR : pourquoi ce code est-il bogué ?

```
int compteur = 100;
while (compteur < 10)
{
    cout << "Le compteur indique : " << compteur;
    compteur--;
}
```

8. CHERCHEZ L'ERREUR : pourquoi ce code est-il bogué ?

```
cout << "Entrez une valeur comprise entre 0 et 5 : ";
cin >> Valeur;
switch (Valeur)
{
   case 0:
         Traitement0();
   case 1:              // case suivant
   case 2:              // case suivant
   case 3:              // case suivant
   case 4:              // case suivant
   case 5:
         Traitement1A5();
         break;
   default:
         ParDefaut();
         break;
}
```

Partie II

La première partie de cet ouvrage vous a initié aux bases de C++. Vous êtes maintenant en mesure de saisir du code, de suivre le déroulement d'une application, de définir des objets et des classes et d'utiliser le compilateur.

Le Chapitre 8 traite des pointeurs. Les pointeurs sont traditionnellement réputés comme l'un des sujets les plus ardus pour les programmeurs débutants. Aussi l'approche est-elle progressive pour vous permettre d'en assimiler parfaitement les concepts et d'écrire vos premiers programmes. Dans le Chapitre 9, vous ferez connaissance avec les références, qui sont des proches cousins des pointeurs. Le Chapitre 10 est consacré à la surcharge de fonctions.

Le Chapitre 11 traite de la conception et de l'analyse orientée objet. Le Chapitre 12 aborde le concept d'héritage, très important en POO, dont l'étude se poursuit au Chapitre 13 qui étudie les tableaux et les collections. Enfin, le Chapitre 14 traite du polymorphisme.



8

Pointeurs

Au sommaire de ce chapitre

- Nature et fonctionnalité des pointeurs
- Déclaration et utilisation de pointeurs
- Espace adressable et gestion de la mémoire

La gestion directe de la mémoire à l'aide de pointeurs constitue une fonctionnalité puissante mais de bas niveau du langage C++. C'est un avantage de C++ par rapport à d'autres langages, comme C# et Visual Basic, même si C# permet d'utiliser les pointeurs sous certaines conditions.

Les pointeurs posent deux problèmes aux programmeurs débutants : d'une part, leur approche est ardue et, d'autre part, il est difficile de déterminer quand et comment il est souhaitable de les utiliser. Dans ce chapitre, les pointeurs sont présentés de façon progressive, de sorte que vous saurez exactement quand y avoir recours.



La possibilité d'utiliser des pointeurs et de manipuler la mémoire à bas niveau est l'un des facteurs qui font de C++ un langage de choix pour les applications embarquées et en temps réel.

Qu'est-ce qu'un pointeur ?

Un pointeur est une variable qui contient une adresse mémoire. C'est tout. Si vous comprenez cette simple phrase, vous savez l'essentiel au sujet des pointeurs.

Mémoire

Pour comprendre les pointeurs, vous devez avoir quelques notions sur l'organisation de la mémoire d'un ordinateur. Cette mémoire est divisée en emplacements numérotés séquentiellement. Chaque variable réside dans un emplacement unique en mémoire, connu par son *adresse*. La Figure 8.1 représente schématiquement la façon dont la variable `unsigned long monAge` est stockée en mémoire.

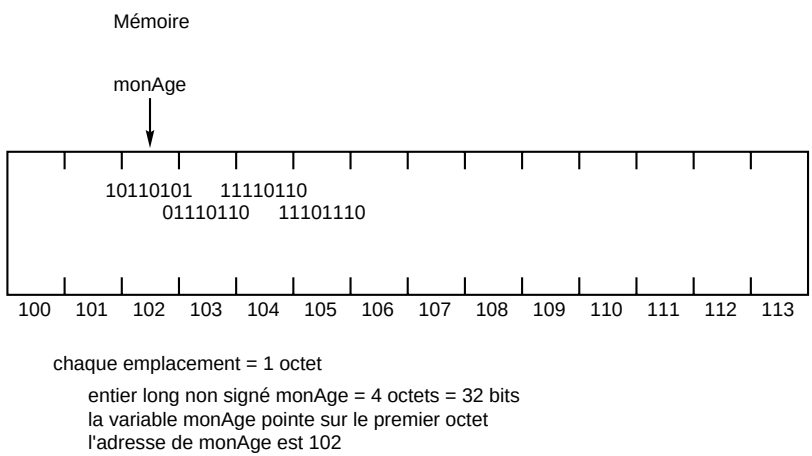


Figure 8.1
Représentation schématique de la variable `monAge`.

Récupérer l'adresse mémoire d'une variable

L'organisation de la mémoire et la numérotation des adresses varient d'un ordinateur à un autre. En général, les développeurs n'ont pas besoin de connaître les adresses des variables utilisées, car ces détails sont gérés par le compilateur. Si vous avez besoin de connaître l'adresse d'un objet en mémoire, vous pouvez toutefois utiliser l'opérateur *adresse de* (`&`), comme le montre le Listing 8.1.

Listing 8.1 : Exemple d'utilisation de l'opérateur *adresse de*

```
 1:  // Listing 8.1 - Utilisation de l'opérateur "adresse de" (&)
 2:  // et adresses des variables locales
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     using namespace std;
 8:     unsigned short varCourte = 5;
 9:     unsigned long  varLongue = 65535;
10:     long varSignee = -65535;
11:
12:     cout << "varCourte :\t" << varCourte;
13:     cout << "\tAdresse de varCourte :\t";
14:     cout <<  &varCourte  << endl;
15:
16:     cout << "varLongue :\t"  << varLongue;
17:     cout  << "\tAdresse de varLongue :\t" ;
18:     cout <<  &varLongue  << endl;
19:
20:     cout << "varSignee :\t"     << varSignee;
21:     cout << "\tAdresse de varSignee :\t" ;
22:     cout <<  &varSignee     << endl;
23:
24:     return 0;
25:  }
```

Ce programme produit le résultat suivant :

```
varCourte :5        Adresse de varCourte :  0xbffff97e
varLongue :65535    Adresse de varLongue :  0xbffff978
varSignee :-65535   Adresse de varSignee :  0xbffff974
```

(les valeurs affichées sur votre ordinateur seront probablement différentes, en particulier dans la dernière colonne).

Le programme commence par la déclaration et l'initialisation de trois variables : la première de type `unsigned short` (ligne 8), la deuxième de type `unsigned long` (ligne 9) et la dernière de type `long` (ligne 10). Leurs valeurs et adresses respectives sont extraites à l'aide de l'opérateur *adresse de* (`&`), puis affichées à l'écran (lignes 12 à 22). Cet opérateur est simplement placé devant le nom de la variable pour que l'adresse soit renvoyée.

La variable `varCourte` est initialisée à la ligne 12. Dans la première ligne du résultat, son adresse hexadécimale est égale à `00bffff97e` lorsque le programme tourne sur un ordinateur

doté d'un processeur Intel Core 2 Duo (il a été ici compilé sur 32 bits). Cette adresse est propre à chaque ordinateur et varie d'une session à une autre. Les résultats que vous obtiendrez seront différents.

Lorsque l'on déclare une variable, le compilateur détermine l'espace mémoire nécessaire en fonction du type de la variable. Il s'occupe ensuite d'allouer automatiquement la mémoire et d'affecter à une adresse pour cette variable. Un entier long occupant généralement 4 octets, le compilateur affectera une adresse qui pointera sur 4 octets en mémoire.

Stockage d'une adresse de variable dans un pointeur

Comme nous l'avons vu, toute variable est associée à une adresse. Il est possible de stocker cette adresse dans un pointeur sans en connaître la valeur.

Supposons que la variable `votreAge` soit de type entier. Le pointeur `pAge` associé, qui va recevoir son adresse, sera déclaré ainsi :

```
int *pAge = 0;
```

Il s'agit d'un *pointeur sur un entier*, ce qui signifie que `pAge` est destiné à recevoir l'adresse d'un entier.

Notez que `pAge` est une variable. Lorsque vous déclarez une variable entière (de type `int`), le compilateur réserve la mémoire nécessaire pour le stockage d'un entier. Si vous déclarez une variable pointeur telle que `pAge`, le compilateur réserve suffisamment de mémoire pour contenir une adresse (4 octets sur la plupart des ordinateurs). Un pointeur, et donc `pAge`, est simplement un type de variable différent.

Noms des pointeurs

Les pointeurs n'étant que des variables, vous pouvez utiliser n'importe quel nom autorisé pour les variables : les mêmes règles et suggestions s'appliquent. De nombreux programmeurs respectent la convention de nommage avec un *p* initial, comme dans `pAge` ou `pNombre`.

Dans l'exemple,

```
int *pAge = 0;
```

`pAge` est initialisé à 0, ce qui caractérise les *pointeurs null*. Les pointeurs devant toujours être initialisés, la solution consiste à affecter une valeur nulle lorsque l'adresse n'est pas encore déterminée. Un pointeur qui n'est pas initialisé est très dangereux, on l'appelle un *pointeur fou*, car il peut pointer vers n'importe quoi.



Pour éviter les erreurs de compilation et d'exécution, n'oubliez pas d'initialiser tous vos pointeurs.

Pour qu'un pointeur stocke une adresse, cette adresse doit lui être affectée. Dans l'exemple précédent, vous devez affecter l'adresse de `MonAge` à `page` :

```
unsigned short int MonAge = 50;  // crée une variable
unsigned short int * pAge = 0;   // crée un pointeur
pAge = &MonAge;             // affecte l'adresse de MonAge à pAge
```

La première ligne initialise à 50 la variable `MonAge`, de type `unsigned short int`. La seconde ligne déclare `pAge` comme un pointeur vers un `unsigned short int` et l'initialise à zéro. On sait que `pAge` est un pointeur grâce à l'astérisque (`*`) placée entre le type et le nom de la variable.

La dernière ligne affecte l'adresse de `MonAge` au pointeur `pAge`, grâce à l'opérateur *adresse de* (`&`). Sans lui, la valeur de `MonAge` aurait été affectée au pointeur `pAge`. Cette valeur aurait pu, ou non, être une adresse valide.

La valeur de l'adresse de `MonAge` est à présent stockée dans `pAge`. `MonAge` contient la valeur 50. L'affectation de ces valeurs aurait pu être simplifiée :

```
unsigned short int monAge = 50;       // variable
unsigned short int * pAge = &monAge;  // pointeur sur monAge
```

Le pointeur `pAge` contient l'adresse de la variable `MonAge`.

Obtenir la valeur d'une variable

À l'aide de `pAge`, vous pouvez déterminer la valeur numérique contenue dans `MonAge`. La technique consistant à accéder à la valeur stockée dans une variable par l'intermédiaire d'un pointeur s'appelle *adressage indirect* ou *indirection*, car vous accédez indirectement à la variable, *via* un pointeur. Vous pouvez, par exemple, utiliser l'indirection avec le pointeur `pAge` pour accéder à la valeur `MonAge`.

L'*indirection* consiste à accéder à la valeur stockée à l'adresse que contient un pointeur. Le pointeur fournit un moyen indirect d'accéder à la valeur stockée à cette adresse.



Avec une variable normale, le type indique au compilateur la taille mémoire nécessaire pour contenir la valeur. Dans le cas d'un pointeur, c'est différent : tous les pointeurs ont la même taille, généralement 4 octets sur un processeur 32 bits et 8 octets sur un processeur 64 bits. Le type indique alors au compilateur la mémoire nécessaire pour contenir l'objet à l'adresse que contient le pointeur !

Dans la déclaration :

```
unsigned short int * pAge = 0;      // création d'un pointeur
```

pAge est déclaré comme un pointeur sur un entier court non signé. Le compilateur sait que le pointeur (qui a besoin de 4 octets pour contenir une adresse) contiendra l'adresse d'un objet du type entier court non signé, qui lui-même nécessite 2 octets.

Déréférencement avec l'opérateur d'indirection

L'opérateur d'indirection (*) est également appelé *opérateur de déréférencement*. Lorsqu'un pointeur est déréférencé, on obtient la valeur située à l'adresse stockée dans le pointeur.

L'accès à la valeur d'une variable classique est direct. Pour créer la variable `VotreAge`, de type entier court non signé et lui affecter la valeur de `MonAge`, il suffit d'écrire les instructions suivantes :

```
unsigned short int VotreAge;
VotreAge = MonAge;
```

Pour affecter la valeur de `MonAge` à la variable `VotreAge`, en utilisant le pointeur `pAge`, vous devez écrire :

```
unsigned short int VotreAge;
VotreAge = *pAge;
```

L'opérateur d'indirection (*) devant une variable pointeur signifie : "valeur stockée à l'adresse". Dans le cas présent, cette affectation signifie donc : "prend la valeur stockée à l'adresse de `pAge`, puis affecte-la à la variable `VotreAge`". Si vous n'aviez pas inclus l'opérateur d'indirection :

```
VotreAge = pAge;  // Mauvais !!
```

vous tenteriez d'affecter la valeur de `pAge`, une adresse mémoire, à `votreAge`. Votre compilateur afficherait probablement un avertissement.

Les différentes utilisations de l'astérisque

L'astérisque (*) est utilisé de deux façons différentes avec les pointeurs : comme composante du pointeur et comme opérateur d'indirection.

Lorsque vous déclarez un pointeur, l'astérisque fait partie de la déclaration et suit le type de l'objet vers lequel on pointe. Exemple :

```
// création d'un pointeur vers un unsigned short
unsigned short * pAge = 0;
```

Par ailleurs, l'opérateur d'indirection (ou de déréférencement) appliqué à un pointeur permet d'accéder non pas à l'adresse elle-même, mais à l'emplacement mémoire situé à cette adresse. Exemple :

```
*pAge = 5;  // affecte 5 à la valeur située à pAge
```

Ce caractère est aussi utilisé comme opérateur de multiplication. Rassurez-vous : le compilateur sait, d'après le contexte, de quel opérateur il s'agit.

Pointeurs, adresses et variables

Avant de continuer, il convient de distinguer le pointeur, l'adresse stockée dans le pointeur et la valeur à l'adresse contenue par le pointeur. Ces trois concepts sont souvent source de confusion chez les programmeurs.

Prenons un exemple :

```
int maVariable = 5;
int * pPointeur = &maVariable ;
```

La variable `maVariable` de type entier reçoit la valeur numérique 5. Le pointeur `pPointeur` est déclaré comme pointeur sur un entier et est initialisé avec l'adresse de `maVariable`. La valeur de l'adresse stockée dans `pPointeur` est l'adresse de `maVariable`. La Figure 8.2 est une représentation graphique de cet exemple.

Dans cette figure, la valeur 5 est stockée à l'adresse mémoire 101, indiquée par le nombre binaire :

```
0000 0000 0000 0101
```

Il s'agit de deux octets (16 bits) dont la valeur décimale est 5.

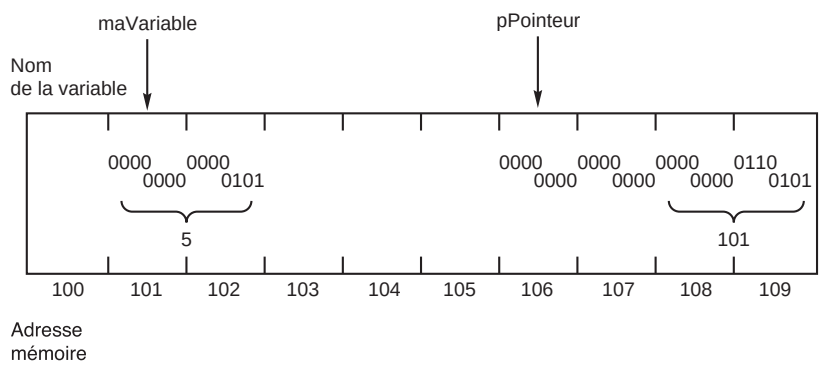


Figure 8.2
Représentation schématique de la mémoire.

La variable pointeur se trouve à l'emplacement 106. Sa valeur est :

```
000 0000 0000 0000 0000 0000 0110 0101
```

Il s'agit de la représentation binaire de la valeur 101, qui est l'adresse de `maVariable` dont la valeur est 5.

Bien que la mémoire soit représentée ici de façon schématique, elle illustre bien la façon dont les pointeurs stockent une adresse.

Gestion de données à l'aide de pointeurs

Outre l'opérateur d'indirection qui permet de lire les données stockées à une position pointée par une variable, vous pouvez aussi manipuler ces données car, après avoir affecté une adresse à un pointeur, vous pouvez avoir accès aux données qu'il pointe.

Le Listing 8.2 reprend ce que vous venez d'apprendre sur les pointeurs. Vous verrez comment l'adresse d'une variable locale est affectée à un pointeur, qui peut ensuite être utilisé avec l'opérateur d'indirection pour manipuler les données contenues dans cette variable.

Listing 8.2 : Manipulation de données par pointeurs

```
 1:  // Listing 8.2 Utilisation des pointeurs
 2:  #include <iostream>

 4:  typedef unsigned short int USHORT;
 5:
 6:  int main()
 7:  {
 8:
 9:     using namespace std;
10:
11:     USHORT monAge;         // une variable
12:     USHORT * pAge = 0;     // un pointeur
13:
14:     moAge = 5;
15:
16:     cout << "moAge : " << moAge << endl;
17:     pAge = &moAge;      // affecte l'adresse de monAge à pAge
18:     cout << "*pAge : " << *pAge << endl << endl;
19:
20:     cout << "Affectation à *pAge de la valeur 7... " << endl;
21:     *pAge = 7;            // affecte la valeur 7 à monAge
22:
23:     cout << "*pAge : " << *pAge << endl;
```

```
24:     cout << "monAge : " << monAge << endl << endl;
25:
26:     cout << "Affectation à moAge de la valeur 9... " << endl;
27:     monAge = 9;
28:
29:     cout << "monAge : " << monAge << endl;
30:     cout << "*pAge : " << *pAge << endl;
31:
32:     return 0;
33: }
```

Ce programme produit le résultat suivant :

```
monAge : 5
*pAge : 5

Affectation à *pAge de la valeur 7...
*pAge : 7
monAge : 7

Affectation à myAge de la valeur 9...
monAge : 9
*pAge : 9
```

Ce programme commence par déclarer deux variables : `monAge` (de type entier court) et son pointeur `pAge`. La ligne 14 affecte 5 à la variable `monAge`, ce que confirme l'affichage produit par la ligne 16.

À la ligne 17, `pAge` reçoit l'adresse de `monAge`. La ligne 18 "déréférence" le pointeur avec l'opérateur d'indirection (`*`) et obtient ainsi la valeur stockée dans `monAge`, c'est-à-dire 5, qui est affichée. La valeur 7 est ensuite affectée à la variable par l'intermédiaire du pointeur (ligne 21). Les messages produits aux lignes 23 et 24 confirment cette opération. Remarquez une fois de plus que l'accès indirect à la variable a été obtenu avec l'astérisque (l'opérateur d'indirection, dans ce contexte).

Enfin, le chiffre 9 est affecté à la variable `monAge` (ligne 27). La ligne 29 lit directement la variable alors que la ligne suivante la lit indirectement, en déférençant `pAge`.

Lecture d'une adresse

Un pointeur permet de manipuler une adresse sans même connaître sa valeur. L'affectation d'une adresse de variable à un pointeur fait que celui-ci contient l'adresse de cette valeur. Nous allons juste le vérifier à l'aide du Listing 8.3.

Listing 8.3 : Lire le contenu d'un pointeur

```
 1:  // Listing 8.3
 2:  Ce qui est stocké dans un pointeur.
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     using namespace std;
 8:
 9:     unsigned short int monAge = 5, tonAge = 10;
10:
11:      // un pointeur
12:     unsigned short int * pAge = &monAge;
13:
14:     cout << "monAge :\t" << monAge
15:        <<  "\t\ttonAge :\t" << tonAge << endl;
16:
17:     cout << "&monAge :\t" << &monAge
18:        << "\t&tonAge :\t" << &tonAge << endl;
19:
20:     cout << "pAge :\t" << pAge << endl;
21:     cout << "*pAge :\t" << *pAge << endl;
22:
23:
24:     cout << "\nRéaffectation de pAge = &tonAge... "
24a!    << endl << endl;
25:     pAge = &tonAge;       // réaffectation du pointeur
26:
27:     cout << "monAge :\t" << monAge <<
28:        "\t\ttonAge :\t" << tonAge << endl;
29:
30:     cout << "&monAge :\t" << &monAge
31:        << "\t&tonAge :\t" << &tonAge << endl;
32:
33:     cout << "pAge :\t" << pAge << endl;
34:     cout << "*pAge :\t" << *pAge << endl;
35:
36:     cout << "\n&pAge :\t" << &pAge << endl;
37:
38:     return 0;
39:  }
```

Ce programme produit le résultat suivant :

```
monAge :    5              tonAge :    10
&monAge :   0xbffff96e     &tonAge :   0xbffff96c
```

```
pAge :      0xbffff96e
*pAge :     5

Réaffectation de pAge = &tonAge...

monAge :    5              tonAge :    10
&monAge :   0xbffff96e     &tonAge :   0xbffff96c
pAge :      0xbffff96c
*pAge :     10

&pAge :     0xbffff968
```

(Le résultat que vous obtiendrez sera différent.)

La ligne 9 déclare les variables entières `monAge` et `tonAge`, de type `unsigned short`, puis déclare le pointeur `pAge` et l'initialise avec l'adresse de `monAge` (ligne 12).

Les valeurs numériques et les adresses des deux variables apparaissent alors à l'écran (lignes 14 à 18). À la ligne 20, vous pouvez remarquer que le *contenu* de `pAge` correspond à l'adresse de `monAge`. La ligne 21 affiche le résultat du déréférencement du pointeur, ce qui affiche la valeur pointée par `pAge`, qui est celle de la variable `monAge`, c'est-à-dire 5.

Cet exemple illustre parfaitement l'extraction de données à l'aide d'un pointeur. Elle est directe à la ligne 20 et indirecte (par déréférencement du pointeur) à la ligne suivante. Avant de poursuivre, vous devez comprendre ces deux approches. Au besoin, modifiez les valeurs et examinez le résultat obtenu.

La ligne 25 affecte l'adresse de `tonAge` au pointeur : `pAge` "pointe" donc désormais vers cette variable. Pour vous aider à comprendre l'opération, les valeurs et les adresses apparaissent de nouveau à l'écran. Le pointeur `pAge` contient désormais l'adresse de `tonAge`. Par déréférencement, on obtient donc la valeur de `tonAge`.

L'adresse du pointeur `pAge` apparaît à l'écran (ligne 36). Comme toute variable, le pointeur possède une adresse (qui peut elle-même être copiée dans un pointeur). Nous présenterons bientôt l'affectation d'une adresse de pointeur à un autre pointeur.

Faire	*Ne pas faire*
● Accéder aux données stockées à l'adresse d'un pointeur, à l'aide de l'opérateur d'indirection (`*`). ● Initialiser l'ensemble des pointeurs d'un programme avec une adresse existante ou la valeur nulle (`0`).	● Confondre l'adresse stockée dans un pointeur avec la valeur stockée à cette adresse.

Utilisation des pointeurs

Pour déclarer un pointeur, vous devez d'abord indiquer le type de la variable ou de l'objet pointé, suivi de l'opérateur * et du nom du pointeur. Exemple :

```
unsigned short int * pPointeur = 0;
```

Pour affecter une valeur à un pointeur ou l'initialiser, vous devez faire précéder le nom de la variable sur laquelle il porte de l'opérateur *adresse de* (&). Exemple :

```
unsigned short int maVariable = 5;
unsigned short int * pPointeur = &maVariable;
```

Pour déréférencer un pointeur, faites précéder son nom de l'opérateur (*). Exemple :

```
unsigned short int maValeur = *pPointeur
```

Utilité des pointeurs

Vous venez d'apprendre les détails de l'affectation d'une adresse de variable à un pointeur. Dans la pratique, cependant, vous ne le ferez jamais. Pourquoi passer par un pointeur pour lire une variable, alors que la valeur de cette dernière est directement accessible ? L'objectif de ce type de manipulation était simplement de vous montrer le fonctionnement des pointeurs. Maintenant que leur syntaxe n'a plus de secrets pour vous, vous pouvez les utiliser correctement. Les pointeurs sont généralement utilisés pour trois raisons :

- gérer des données dans l'espace mémoire adressable ;
- accéder aux données membres et aux fonctions des classes ;
- passer des variables par référence à des fonctions.

Le reste de ce chapitre est consacré à la gestion de données dans l'espace mémoire adressable et à l'accès aux données membres et aux fonctions d'une classe. Le passage de variables à l'aide des pointeurs, appelé passage par référence, sera étudié au chapitre suivant.

La pile et l'espace mémoire adressable (tas)

À la fin du Chapitre 5, nous avons évoqué les cinq parties suivantes de la mémoire :

- l'espace global de noms ;
- l'espace mémoire adressable (appelé aussi mémoire "heap" ou *tas*) ;
- les registres ;

- l'espace de code ;
- la pile.

Les variables locales et les paramètres passés aux fonctions sont placés sur *la pile*. Le code du programme réside dans l'*espace de code*. Quant aux *registres*, ils permettent au système de réaliser des opérations internes, comme le suivi du contenu de la pile et du pointeur d'instruction. Le reste de la mémoire est essentiellement composé de l'espace adressable, aussi appelé le *tas*.

Les variables locales disparaissent lorsque la fonction qui les a définies se termine : le programmeur n'a donc pas besoin de gérer cette partie de la mémoire. Par contre, ces objets locaux ne peuvent pas être facilement transmis à d'autres objets ou à d'autres fonctions : pour ce faire, il faut copier ces objets de la pile vers la valeur de retour de la fonction, puis vers l'objet destinataire dans l'appelant et cela a un coût. Les variables globales permettent de pallier cet inconvénient au prix d'un accès illimité aux données, ce qui peut rendre le code illisible et impossible à mettre à jour. La solution consiste alors à utiliser le tas, mais le programmeur doit alors gérer correctement ces données.

Le tas peut être considéré comme un ensemble de milliers de cases mémoires destinées à recevoir des données. À la différence de la pile, ces cases ne sont être nommées : pour accéder à une donnée stockée dans le tas, il faut obtenir l'adresse de l'emplacement que l'on a réservé pour elle, puis copier cette adresse dans un pointeur, afin de la mémoriser.

Prenons un exemple. Un ami vous communique par écrit le numéro de téléphone du garage SuperAuto. De retour à votre domicile, vous composez ce numéro, puis vous jetez le papier à la corbeille. Vous ne pouvez donc plus connaître le numéro de téléphone de cette entreprise, mais la mémoire de l'appareil téléphonique continue de vous permettre de les appeler. SuperAuto correspond aux données stockées dans le tas. Vous ne savez pas où le garage se trouve, mais vous savez comment y accéder, en utilisant son adresse – son numéro de téléphone stocké dans la mémoire de votre combiné, ici. Vous ne connaissez même pas ce numéro, mais vous l'avez mis dans un pointeur (la mémoire du combiné), qui vous permet de continuer à joindre le garage. De la même façon, dans un programme, un pointeur permet d'avoir accès à une donnée sans connaître son emplacement en mémoire.

La pile est vidée automatiquement quand la fonction prend fin : toutes les variables locales deviennent hors de portée et sont supprimées de la pile. En revanche, le tas n'est vidé qu'après la fin du programme. Pendant la session du programme, la gestion du tas et la libération de la mémoire vous incombent. C'est dans ce cas que les destructeurs sont absolument nécessaires : ils fournissent un endroit permettant de libérer les emplacements du tas qui ont été alloués lors de la construction de l'objet.

L'avantage du tas est que les emplacements que vous avez réservés restent disponibles jusqu'à ce que vous les libériez explicitement. Si vous allouez de la mémoire sur le tas dans une fonction, cette mémoire reste allouée après la fin de la fonction.

Cette persistance des emplacements réservés est aussi un inconvénient car, si vous oubliez de les libérer, vous pouvez vous retrouver dans une situation où il n'y aura plus assez de place disponible, auquel cas votre système peut se planter.

L'avantage d'accéder à la mémoire de cette façon au lieu de passer par des variables globales est que seules les fonctions qui ont accès au pointeur (qui contient l'adresse appropriée) peuvent accéder aux données sur le tas. Cela impose donc que l'objet contenant le pointeur, ou le pointeur lui-même, soit explicitement passé à toute fonction apportant des changements, ce qui réduit les risques qu'une fonction puisse subrepticement modifier des données.

Pour que tout cela fonctionne, vous devez donc créer un pointeur vers une zone du tas, puis le passer comme paramètre aux fonctions, comme on l'explique dans la section qui suit.

Allouer de l'espace avec le mot-clé *new*

Le mot-clé `new` permet de réserver de l'espace dans le tas. Il doit être suivi du type de l'objet afin que le compilateur détermine la quantité de mémoire requise. Ainsi, l'expression `new unsigned short int` et l'expression `new long` réservent, respectivement, 2 et 4 octets sur le tas, en supposant que votre système utilise un entier court non signé sur 2 octets et une entier long sur 4 octets.

La valeur renvoyée par `new` est l'adresse de l'emplacement mémoire réservé sur le tas. Sachant que les adresses mémoire sont stockées dans des pointeurs, il n'est pas surprenant que la valeur de retour de `new` soit affectée à un pointeur. Pour créer un entier court non signé dans le tas, on écrira donc :

```
unsigned short int * pPointeur;
pPointeur = new unsigned short int;
```

Bien entendu, on peut faire tout cela en une seule ligne, en initialisant le pointeur lors de sa déclaration :

```
unsigned short int * pPointeur = new unsigned short int;
```

Dans les deux cas, `pPointeur` pointera sur une variable de type entier court non signé résidant dans le tas. Vous pouvez également placer une valeur dans cette zone mémoire :

```
*pPointeur = 72;
```

Cette instruction signifie "Affecter 72 à la valeur pointée par `pPointeur`," ou "Affecter la valeur 72 à l'emplacement du tas sur lequel `pPointeur` pointe".



S'il ne parvient pas à réserver de l'espace supplémentaire dans le tas (la mémoire est une ressource limitée), `new` lance une exception (voir Chapitre 20).

Libérer de la mémoire : le mot-clé *delete*

Après utilisation, la zone de mémoire définie à l'aide du mot-clé `new` doit être libérée pour être rendue au système. Pour cela, appelez le mot-clé `delete` sur le pointeur.

Il est essentiel de se souvenir que la mémoire allouée grâce à `new` n'est pas automatiquement libérée. Si une variable de pointeur pointe vers de la mémoire dans l'espace libre et que le pointeur devient hors de portée, la mémoire n'est pas automatiquement rendue à l'espace libre. Elle est considérée allouée ; le pointeur n'étant plus disponible, vous ne pouvez plus y accéder. Cela survient, par exemple, lorsqu'un pointeur est une variable locale. Au retour de la fonction dans laquelle ce pointeur est déclaré, la portée du pointeur cesse et celui-ci est perdu. La mémoire allouée avec `new` n'est pas libérée (et ne peut plus l'être puisqu'on a perdu le pointeur) – elle devient indisponible.

Cette situation est appelée *fuite mémoire*, car l'espace mémoire ne pourra être récupéré qu'après la fin du programme : c'est comme si la mémoire fuyait de votre ordinateur.

Pour éviter ces fuites de mémoire, vous devrez restituer au tas toute la mémoire que vous avez allouée en utilisant le mot-clé `delete` :

```
delete pPointeur;
```

Cette opération permet de libérer la mémoire dont l'adresse est stockée dans le pointeur. Cela revient donc à dire "libère l'emplacement du tas pointé par ce pointeur". Le pointeur reste un pointeur et peut être réaffecté. Dans le Listing 8.4, nous effectuerons trois opérations : définir une variable et l'allouer dans le tas, l'utiliser puis la supprimer.

Le plus souvent, on alloue des éléments sur le tas à partir d'un constructeur et on les libère dans le destructeur. Dans d'autres cas, on initialise les pointeurs dans le constructeur, on alloue les emplacements à mesure que l'objet est utilisé et, dans le destructeur, on vérifie si ces pointeurs sont nuls et on libère les emplacements qu'ils pointent si ce n'est pas le cas.



Lorsque vous appelez `delete` sur un pointeur, vous libérez la mémoire sur laquelle il pointe. Vous ne devez le faire qu'une seule fois, sous peine d'entraîner une erreur d'exécution. Pour éviter ce désagrément, affectez la valeur nulle (`0`) au pointeur aussitôt après l'avoir supprimé. Appeler `delete` sur un

pointeur nul n'aura aucun incidence sur le déroulement du programme. Voici un exemple :

```
Animal *pChien = new Animal;   // allouer de la mémoire
delete pChien; // libère la mémoire
pChien = 0; // affecte la valeur nulle au pointeur
//...
delete pChien; // sans dommage pour le programme
```

Listing 8.4 : Allocation, utilisation et suppression d'un pointeur

```
 1:  // Listing 8.4
 2:  // Allocation et suppression d'un pointeur
 3:  #include <iostream>
 4:  int main()
 5:  {
 6:     using namespace std;
 7:     int variableLocale = 5;
 8:     int * pLocal= &variableLocale;
 9:     int * pTas = new int;
10:     *pTas = 7;
11:     cout << "variableLocale : " << variableLocale << endl;
12:     cout << "*pLocal : " << *pLocal << endl;
13:     cout << "*pTas : " << *pTas << endl;
14:     delete pTas;
15:     pTas = new int;
16:     *pTas = 9;
17:     cout << "*pTas : " << *pTas << endl;
18:     delete pTas;
19:     return 0;
20:  }
```

Ce programme produit le résultat suivant :

```
variableLocale : 5
*pLocal : 5
*pTas : 7
*pTas : 9
```

La ligne 7 déclare et initialise la variable locale (ironiquement nommée `variableLocale`). La ligne 8 déclare un pointeur appelé `pLocal` et l'initialise avec l'adresse de la variable locale. La ligne 9 déclare un second pointeur, appelé `pTas` ; comme elle l'initialise avec la valeur renvoyée par l'instruction `new int`, il désigne donc un nouvel emplacement réservé pour un entier sur le tas. La valeur 7 est affectée à l'espace mémoire qui vient d'être réservé (ligne 10).

La ligne 11 affiche la valeur de la variable locale (`variableLocale`), la ligne 12 celle de l'emplacement pointé par `pLocal` et la ligne 13, la valeur de l'emplacement pointé par `pTas`, On remarque que les valeurs affichées par les lignes 11 et 12 se correspondent.

La ligne 14 libère avec `delete` l'espace mémoire qui a été alloué à la ligne 9. Le pointeur est donc dissocié de l'adresse sur laquelle il pointait et peut maintenant pointer vers un autre emplacement mémoire. On le réaffecte aux lignes 15 et 16 et on affiche la valeur du nouvel emplacement qu'il pointe à la ligne 17. Enfin, la ligne 18 libère à nouveau l'emplacement qui a été alloué.

Bien que la ligne 18 soit redondante car la mémoire sera de toute façon libérée à la fin du programme, il est préférable de libérer explicitement cette mémoire. Si le programme est ensuite modifié ou étendu, il sera toujours bénéfique d'avoir déjà pris en compte cette étape.

En savoir plus sur les fuites mémoire

Les fuites mémoire sont l'un des problèmes les plus sérieux des pointeurs. Nous venons de voir une des situations qui provoquent ces fuites, mais réaffecter une nouvelle adresse à un pointeur avant de libérer l'emplacement qu'il pointait en est une autre. Voici un exemple :

```
1:   unsigned short int * pPointeur = new unsigned short int;
2:   *pPointeur = 72;
3:   pPointeur = new unsigned short int;
4:   *pPointeur = 84;
```

La ligne 1 crée `pPointeur` et lui affecte l'adresse d'un espace mémoire libre sur le tas. La ligne 2 y stocke la valeur `72`. La ligne 3 réaffecte `pPointeur` à un autre emplacement mémoire et la ligne 4 y place la valeur `84`. L'emplacement d'origine – qui contient `72` – est donc désormais indisponible puisque le pointeur vers cet emplacement a été réaffecté : il n'est plus possible d'accéder à cet emplacement et on ne peut plus non plus le libérer avant la fin du programme.

Il aurait fallu utiliser le code suivant :

```
1: unsigned short int * pPointeur = new unsigned short int;
2: *pPointeur = 72;
3: delete pPointeur;
4: pPointeur = new unsigned short int;
5: *pPointeur = 84;
```

La mémoire sur laquelle pointait à l'origine `pPointeur` est supprimée et par conséquent libérée à la ligne 3.



À chaque instruction `new` devrait correspondre une instruction `delete`. En effet, il est important de libérer l'espace mémoire dès qu'il n'est plus nécessaire.

Création d'objets sur le tas

Un pointeur peut pointer sur n'importe quel type de données, y compris des instances de classes. Si vous disposez d'une classe `Chat`, par exemple, vous pouvez déclarer un pointeur permettant de pointer des objets de cette classe et créer une instance de `Chat` sur le tas, exactement comme vous pouvez créer un objet sur la pile. La syntaxe est la même que pour les entiers :

```
Chat *pChat = new Chat;
```

Cette instruction appelle le constructeur par défaut (c'est-à-dire le constructeur qui ne prend pas de paramètre). Le constructeur s'exécute dès qu'un objet est créé (que ce soit sur la pile ou sur le tas). Vous n'êtes pas limité au constructeur par défaut lors de la création d'un objet avec `new` : vous pouvez employer n'importe quel constructeur.

Suppression d'objets du tas

L'instruction `delete` appelle le destructeur de l'objet avant de libérer l'emplacement pointé, ce qui permet d'effectuer des opérations de nettoyage (consistant, généralement, à libérer d'autres emplacements du tas, alloués par l'objet), exactement comme pour un objet alloué sur la pile. Le Listing 8.5 crée et supprime des objets sur le tas.

Listing 8.5 : Création et suppression d'objets sur le tas

```
 1:  // Listing 8.5 - Création d'objets dans le tas
 2:  // à l'aide de new et delete
 3:
 4:  #include <iostream>
 5:
 6:  using namespace std;
 7:
 8:  class ChatSimple
 9:  {
10:  public:
11:     ChatSimple();
12:     ~ChatSimple();
13:  private:
14:     int sonAge;
```

```
15:  };
16:
17:  ChatSimple::ChatSimple ()
18:  {
19:     cout << "Appel du constructeur." << endl;
20:     sonAge = 1;
21:  }
22:
23:  ChatSimple::~ChatSimple ()
24:  {
25:     cout << "Appel du destructeur." << endl;
26:  }
27:
28:  int main()
29:  {
30:     cout << "ChatSimple Frisky... " << endl;
31:     ChatSimple Frisky;
32:     cout << "ChatSimple *pRags = new ChatSimple..." << endl;
33:     ChatSimple * pRags = new ChatSimple;
34:     cout << "Suppression de pRags... " << endl;
35:     delete pRags;
36:     cout << "Fin, Frisky disparait... " << endl;
37:     return 0;
38:  }
```

Ce programme produit le résultat suivant :

```
ChatSimple Frisky...
Appel du constructeur.
ChatSimple *pRags = new ChatSimple...
Appel du constructeur.
Suppression de pRags...
Appel du destructeur.
Fin, Frisky disparait...
Appel du destructeur.
```

Les lignes 8 à 15 déclarent une classe `ChatSimple` minimale. La ligne 11 déclare son constructeur, qui est défini par les lignes 17 à 21. La ligne 12 déclare son destructeur, qui est défini par les lignes 23 à 26. Le constructeur et le destructeur affichent un message simple indiquant qu'ils ont été appelés.

La ligne 31 crée `Frisky` comme une variable locale ordinaire, donc sur la pile. Cette création fait appel au constructeur. Deux lignes plus loin, on crée également un objet `ChatSimple`, qui sera pointé par `pRags` ; cette fois-ci, puisque nous utilisons un pointeur, cet objet sera crée sur le tas. Là aussi, cette création fait appel au constructeur.

La ligne 35 appelle `delete` sur le pointeur `pRags` ; le destructeur est alors appelé et la mémoire qui a été allouée pour contenir cet objet est libérée. Lorsque la fonction prend fin à la ligne 38, l'objet `Frisky` devient hors de portée et il est supprimé automatiquement à l'aide du destructeur.

Accès aux données membres

Le Chapitre 6 vous a montré comment accéder aux données et aux fonctions membres à l'aide de l'opérateur point (`.`). Les objets de la classe `Chat` avaient été créés sur la pile.

Accéder aux membres d'un objet lors de l'utilisation d'un pointeur est un peu plus complexe. Pour lire l'un de ses membres, vous devez d'abord déréférencer le pointeur, puis utiliser l'opérateur point sur l'objet désigné par le pointeur. Il n'est pas inutile de le répéter : vous devez *d'abord* déréférencer le pointeur et vous utilisez *ensuite* la valeur déréférencée (celle qui est pointée) avec l'opérateur point pour accéder aux membres de l'objet. Pour accéder à la fonction membre `GetAge()` d'un objet pointé par `pRags`, par exemple, vous devez donc utiliser l'instruction suivante :

```
(*pRags).GetAge();
```

Les parenthèses servent à garantir que `pRags` sera déréférencé *avant* l'accès à la fonction `GetAge()` car les parenthèses ont la priorité sur tous les autres opérateurs, y compris le point.

Cette syntaxe pouvant devenir fastidieuse, C++ fournit un raccourci ce genre d'accès indirect : l'opérateur *d'accès au membre d'une classe* (`->`). Cet opérateur se compose de deux caractères : un tiret court (-) et le signe *supérieur à* (>). Dans le Listing 8.6, le programme utilise cet opérateur pour accéder aux données et aux fonctions membres des objets créés sur le tas.



*L'opérateur d'accès au membre d'une classe (`->`) pouvant aussi être employé pour l'accès indirect aux membres d'un objet (*via *un pointeur), il peut également être considéré comme un opérateur d'indirection. Certains programmeurs l'appellent d'ailleurs opérateur pointer sur car c'est exactement ce qu'il fait.*

Listing 8.6 : Lecture de données membres des objets dans le tas

```
1:  // Listing 8.6 - Accès aux données membres des objets du tas
2:  // à l'aide de l'opérateur ->
3:
4:  #include <iostream>
5:
```

```
 6:  class SimpleChat
 7:  {
 8:  public:
 9:     ChatSimple() {sonAge = 2; }
10:     ~ChatSimple() {}
11:     int GetAge() const { return sonAge; }
12:     void SetAge(int age) { sonAge = age; }
13:  private:
14:     int sonAge;
15:  };
16:
17:  int main()
18:  {
19:     using namespace std;
20:     ChatSimple * Frisky = new ChatSimple;
21:     cout << "Frisky a " << Frisky->GetAge() << " ans\n";
22:     Frisky->SetAge(5);
23:     cout << "Frisky a " << Frisky->GetAge() << " ans\n";
24:     delete Frisky;
25:     return 0;
26:  }
```

Ce programme produit le résultat suivant :

```
Frisky a 2 ans
Frisky a 5 ans
```

La ligne 20 crée une instance de `ChatSimple` sur le tas, désignée par le pointeur `Frisky`. Le constructeur par défaut de l'objet définit l'âge à 2. La méthode `GetAge()` extrait la valeur du pointeur, à l'aide de l'opérateur `->` (ligne 21). `Frisky` étant un pointeur, on utilise l'opérateur d'indirection `->` pour accéder à ses données et à ses fonctions membres. La ligne 22 appelle la méthode `SetAge()` et la ligne suivante appelle à nouveau `GetAge()`.

Création de données membres dans le tas

Outre créer des objets sur le tas, vous pouvez aussi créer des données membres sur le tas car des membres d'une classe peuvent eux-mêmes être des pointeurs. Grâce à ce que vous avez appris, vous pouvez allouer de la mémoire sur le tas pour ces pointeurs. Cette allocation peut avoir lieu dans le constructeur de la classe ou dans l'une de ses méthodes. Lorsque vous avez terminé d'utiliser le membre, vous pouvez – et devez – le supprimer dans l'une des méthodes ou dans le destructeur, comme le montre le Listing 8.7.

Listing 8.7 : Les pointeurs comme données membres d'une classe

```
// Listing 8.7 - Données membres pointeurs
// Accès à l'aide de l'opérateur ->

#include <iostream>

class ChatSimple
{
public:
   ChatSimple();
   ~ChatSimple();
   int GetAge() const { return *sonAge; }
   void SetAge(int age) { *sonAge = age; }

   int GetPoids() const { return *sonPoids; }
   void SetPoids (int poids) { *sonPoids = poids; }

private:
   int * sonAge;
   int * sonPoids;
};

ChatSimple::ChatSimple()
{
   sonAge = new int(2);
   sonPoids = new int(5);
}

ChatSimple::~ChatSimple()
{
   delete sonAge;
   delete sonPoids;
}

int main()
{
   using namespace std;
   ChatSimple *Frisky = new ChatSimple;
   cout << "Frisky a " << Frisky->GetAge()
            << " ans " << endl;
   Frisky->SetAge(5);
   cout << "Frisky a " << Frisky->GetAge()
            << " ans\ " << endl;
   delete Frisky;
   return 0;
}
```

Ce programme produit le résultat suivant :

```
Frisky a 2 ans
Frisky a 5 ans
```

La classe `ChatSimple` contient deux variables membres, qui sont toutes les deux des pointeurs vers des entiers (lignes 18 et 19). Le constructeur les initialise afin qu'ils pointent sur des emplacements du tas et affecte des valeurs par défaut à ces emplacements (lignes 22 à 26).

Les lignes 24 et 25 utilisent un pseudo-constructeur sur le nouvel entier, en lui passant une valeur initiale ; cela a pour effet de créer un entier sur le tas et d'initialiser sa valeur (à 2 à la ligne 24, et à 5 à la ligne 25).

Le destructeur libère ensuite la mémoire (lignes 28 à 32). Il n'y a pas besoin d'affecter 0 à ces pointeurs après la libération des zones qu'ils pointaient puisqu'on est dans le destructeur. C'est l'un des cas où il est possible de transgresser la règle que nous avions mentionnées.

La fonction appelante (`main()`, ici) ignore que les variables `sonAge` et `sonPoids` sont des pointeurs ; elle se contente d'appeler les méthodes `GetAge()` et `SetAge()` sans connaître les détails de la gestion mémoire car celle-ci est cachée dans l'implémentation de la classe, ce qui est une bonne chose.

Lorsque `Frisky` est supprimé à la ligne 41, son destructeur est appelé. Il libère alors les emplacements pointés par les deux membres. Si ces derniers pointaient sur des objets de classes définies par l'utilisateur, leurs destructeurs s'exécuteraient également.

Comprendre ce que l'on fait

Utiliser des pointeurs comme dans le Listing 8.7 serait assez ridicule dans un vrai programme, sauf s'il y a de bonnes raisons pour que les objets `Chat` stockent leurs membres de cette façon. Ici, nous n'avions aucune raison d'utiliser des pointeurs pour accéder à `sonAge` et `sonPoids`, mais cela pourrait être très utile dans d'autres cas.

Ceci implique une question évidente : que voulons-nous faire exactement lorsque l'on utilise des pointeurs pour désigner des variables au lieu de variables normales ? Ceci démontre également toute l'importance de la conception. Si l'on a conçu un objet qui fait référence à un autre objet et que le second objet peut naître avant le premier et continuer d'exister après lui, le premier objet doit contenir une référence au second.

Le premier objet pourrait, par exemple, être une fenêtre et le second un document. La fenêtre doit pouvoir accéder au document, mais elle ne contrôle pas la durée de vie du document. Elle doit, par conséquent, contenir une référence au document.

Cette implémentation est réalisée en C++ à l'aide de pointeurs ou de références. Ces dernières seront étudiées au Chapitre 9.

Le pointeur *this*

Toute méthode membre d'une classe a un paramètre caché : le pointeur `this`, qui pointe sur l'objet sur laquelle elle est appelée. Lorsque les fonctions `GetAge()` ou `SetAge()` sont appelées sur un objet particulier, elles reçoivent en paramètre et en coulisses le pointeur `this` de cet objet

Il est possible d'utiliser le pointeur `this` de façon explicite, comme dans le Listing 8.8.

Listing 8.8 : Utilisation du pointeur *this*

```
// Listing 8.8
// Utilisation du pointeur this

#include <iostream>

class Rectangle
{
public:
   Rectangle();
   ~Rectangle();
   void SetLongueur(int longueur)
      { this->saLongueur = longueur; }
   int GetLongueur() const
      { return this->saLongueur; }

   void SetLargeur(int largeur)
      { saLargeur = largeur; }
   int GetLargeur() const
      { return saLargeur; }

private:
   int saLongueur;
   int saLargeur;
};

Rectangle::Rectangle()
{
   saLargeur = 5;
   saLongueur = 10;
}
Rectangle::~Rectangle()
{}

int main()
{
```

```
36:     using namespace std;
37:     Rectangle unRect;
38:     cout << "unRect mesure " << unRect.LireLongueur()
39:        << " cm de long. " endl;
40:     cout << "unRect mesure " << unRect.LireLargeur()
41:        << " cm de large. " << endl;
42:     unRect.SetLongueur(20);
43:     unRect.SetLargeur(10);
44:     cout << "unRect mesure " << unRect.LireLongueur()
45:        << " cm de long. " << endl;
46:     cout << "unRect mesure " << unRect.LireLargeur()
47:        << " cm de large. " << endl;
48:     return 0;
49:  }
```

Ce programme produit le résultat suivant :

```
unRect mesure 10 cm de long.
unRect mesure 5 cm de large.
unRect mesure 20 cm de long.
unRect mesure 10 cm de large.
```

Les fonctions d'accès `SetLongueur()` et `GetLongueur()` [lignes 11-12 et 13-14] utilisent toutes les deux explicitement le pointeur `this` pour accéder aux variables membres de l'objet `Rectangle`, ce qui n'est pas le cas de `SetLargeur()` et `GetLargeur`() [lignes 16 à 19]. Leur comportement est pourtant identique aux deux premières et la syntaxe des instructions est plus simple à comprendre.

S'il ne servait qu'à ça, le pointeur `this` aurait peu d'intérêt, mais c'est en réalité un outil puissant car, en tant que pointeur, il mémorise l'adresse d'un objet.

Le Chapitre 10 présentera un cas d'utilisation de `this`, lorsque nous aborderons la surcharge des opérateurs. Pour le moment, vous pouvez vous contenter de savoir qu'il pointe sur l'objet sur lequel a été appelé la méthode.

Vous n'avez pas à vous soucier de sa création ou de sa suppression : le compilateur s'en charge pour vous.

Pointeurs perdus, incontrôlables ou pointeurs fous

Une fois de plus, les problèmes de pointeurs refont surface. Ceci est dû au fait que les erreurs imputables aux pointeurs sont parmi les plus difficiles à déceler et les plus problématiques. Les pointeurs sont la cause de nombreux bogues particulièrement difficiles à détecter. Un pointeur perdu apparaît lorsque vous le supprimez à l'aide du mot-clé `delete` – vous libérez donc la mémoire vers laquelle il pointait – sans lui affecter ensuite la valeur `null`.

Si vous tentez de l'utiliser à nouveau sans l'avoir réaffecté, le résultat est imprévisible et, dans le meilleur des cas, le programme plantera.

C'est comme si le garage SuperAuto avait déménagé mais que vous continuiez à utiliser le numéro de téléphone que vous aviez mémorisé dans votre combiné. Cela pourrait ne pas avoir de conséquences fâcheuses – un téléphone sonnerait dans une pièce vide. En revanche, si ce numéro a été réaffecté à une usine de munitions, votre appel pourrait déclencher une explosion et souffler toute la ville !

En résumé, faites attention à ne pas utiliser un pointeur qui a été supprimé à l'aide de `delete`. Il pointe toujours sur l'ancienne zone de mémoire, mais le compilateur a pu y mettre d'autres données ; réutiliser ce pointeur sans lui réallouer un nouvel emplacement mémoire peut provoquer le plantage de votre programme. Pire encore, le programme peut continuer tranquillement son exécution et se planter quelques minutes plus tard. Ce type d'erreur est une véritable bombe à retardement. Pour plus de sécurité, affectez la valeur nulle (`0`) à tous les pointeurs que vous supprimez afin de désamorcer ces bombes.



En jargon informatique, les pointeurs perdus sont parfois appelés pointeurs fous, pointeurs incontrôlables ou pointeurs pendants.

Le Listing 8.9 illustre la création d'un pointeur perdu.



N'exécutez pas ce programme ! Il crée délibérément un pointeur perdu. Dans le meilleur des cas, il provoquera un plantage !

Listing 8.9 : Création d'un pointeur perdu

```
 1:  // Listing 8.9 - Un pointeur perdu
 2:
 3:  typedef unsigned short int USHORT;
 4:  #include <iostream>
 5:
 6:  int main()
 7:  {
 8:     USHORT * pInt = new USHORT;
 9:     *pInt = 10;
10:     std::cout << "*pInt : " << *pInt << std::endl;
11:     delete pInt;
12:
13:     long * pLong = new long;
14:     *pLong = 90000;
```

```
15:     std::cout << "*pLong : " << *pLong << std::endl;
16:
17:     *pInt = 20;      // Attention, a été supprimé !
18:
19:     std::cout << "*pInt : " << *pInt  << std::endl;
20:     std::cout << "*pLong : " << *pLong  << std::endl;
21:     delete pLong;
22:     return 0;
23: }
```

Ce programme produit le résultat suivant :

```
*pInt :   10
*pLong :  90000
*pInt :    20
*pLong :   65556
```

(ne tentez pas d'exécuter ce programme. Le résultat que vous obtiendriez serait au mieux différent ; au pire, votre ordinateur planterait).

La ligne 8 définit `pInt` comme un pointeur sur un `USHORT`. À la ligne suivante, la valeur 10 est placée dans la mémoire allouée à `pInt`. La valeur désignée s'affiche alors (ligne 10). L'instruction `delete` supprime ensuite l'emplacement pointé par `pInt`, qui devient alors un pointeur perdu (après la ligne 11).

La ligne 13 déclare un nouveau pointeur, `pLong`, et le fait pointer sur un nouvel emplacement à l'aide de `new`. Cet emplacement la valeur 90 000 en ligne 14 et la ligne suivant affiche cette valeur.

C'est à la ligne 17 que les problèmes commencent. La ligne 17 affecte la valeur 20 à l'emplacement pointé par `pInt` alors que ce dernier ne pointe plus sur une adresse valide puisque la zone mémoire vers laquelle il pointait a été libérée en ligne 11. Cette opération est la porte ouverte provoquera sans aucun doute un désastre.

La ligne 19 affiche le contenu pointé par `pInt`,, c'est-à-dire 20. La ligne suivante affiche le contenu pointé par `pLong` et l'on s'aperçoit qu'il a soudainement été changé en 65 556. Ceci nous amène à poser deux questions :

1. Pourquoi le contenu de `pLong` a-t-il changé spontanément alors qu'on ne l'a pas touché ?
2. Ou a été le nombre 20 utilisé avec `pInt` à la ligne 17 ?

Comme vous pouvez le deviner, ces deux questions sont étroitement liées. La ligne 17 a affecté 20 à la zone de mémoire sur laquelle pointait `pInt` au début du programme. Comme elle a été libérée à la ligne 11, le compilateur a pu la réutiliser. Lorsque le pointeur `pLong` a été créé à la ligne 13, il a été associé à l'ancienne zone mémoire pointée par `pInt`

(sur certaines machines, il est possible que vous ne remarquiez rien selon les emplacements où ont été stockées ces valeurs). Lorsque la valeur 20 a été affectée à l'emplacement anciennement pointé par `pInt`, elle a donc écrasé la valeur pointée par `pLong`. On dit que le pointeur a été "écrasé" et c'est l'une des conséquences possibles de l'utilisation d'un pointeur perdu.

Il s'agit d'un bogue particulièrement vicieux car la valeur qui a été modifiée n'était pas associée au pointeur perdu. Cette modification de la valeur pointée par `pLong` est un effet de bord de la mauvaise utilisation de `pInt`. Dans un programme très long, une telle erreur serait très difficile à détecter et à corriger.

Pour la petite histoire, voici pourquoi l'adresse mémoire de `pLong` contient le nombre 65 556 (Listing 8.9).

1. Alors que `pInt` pointait sur une adresse mémoire particulière, le nombre 10 lui a été affecté.
2. L'instruction `delete` appelée sur `pInt` a indiqué au compilateur qu'il pouvait réutiliser cet emplacement. Il l'a affecté au pointeur `pLong`.
3. La valeur 90 000 a été affectée à `*pLong`. Sur l'ordinateur utilisé, cette valeur de type `long` de 4 octets (`00 01 5F 90`) est stockée dans l'ordre inverse, en permutant les octets. Elle a donc été stockée comme `5F 90 00 01`.
4. `pInt` a été initialisé à 20 (soit `00 14` en notation hexadécimale). Comme il pointait sur la même adresse, les deux premiers octets de `pLong` ont été écrasés. Le contenu de ce dernier est donc devenu `00 14 00 01`.
5. Au moment de l'affichage de cette valeur, le système a effectué une permutation d'octets pour retrouver l'ordre correct `00 01 00 14`, qui a été traduit dans sa valeur décimale, 65556.

Quelle est la différence entre un pointeur nul et un pointeur perdu ?

Lorsque vous supprimez un pointeur, vous demandez au compilateur de libérer la mémoire, mais le pointeur lui-même existe toujours. Il est devenu un pointeur perdu.

Si vous écrivez alors :

```
monPtr = 0;
```

de pointeur perdu, il devient pointeur nul.

Normalement, si vous supprimez un pointeur et que vous le supprimez à nouveau, votre programme est dans un état indéfini, c'est-à-dire que tout peut arriver. Si vous avez de la chance, le programme plantera. Si vous supprimez un pointeur nul, aucune conséquence fâcheuse n'en résultera.

L'utilisation d'un pointeur nul ou perdu (en écrivant, par exemple `monPtr = 5;`) est illégal et peut provoquer un plantage. Avec un pointeur nul, le plantage est certain, ce qui est un autre avantage des pointeurs nuls par rapport aux pointeurs perdus car les plantages prévisibles sont plus faciles à déboguer.

Pointeurs *const*

Vous pouvez utiliser le mot-clé `const` avec les pointeurs en le plaçant avant et/ou après le type pointé. Les déclarations suivantes, par exemple, sont toutes correctes :

```
const int * pUn;
int * const pDeux;
const int * const pTrois;
```

Chacune effectue toutefois des actions différentes :

- `pUn` est un pointeur sur un entier constant. La valeur sur laquelle il pointe ne peut pas être modifiée.
- `pDeux` est un pointeur constant sur un entier. La valeur sur laquelle il pointe peut être modifiée, mais `pDeux` est constant et ne peut donc pas pointer sur un autre emplacement.
- `pTrois` est un pointeur constant sur un entier constant. La valeur sur laquelle il pointe ne peut pas être modifiée et il ne peut pas pointer sur un autre emplacement.

Le mot-clé `const` s'applique à l'élément qui figure immédiatement après lui. Si le type suit le mot-clé `const`, la valeur est constante. Si la variable se trouve à droite du mot-clé `const`, c'est la variable pointeur elle-même qui est constante.

```
const int * p1;  // l'entier pointé est constant

int * const p2;  // le pointeur p2 est constant :
                 // il ne peut pas pointer sur autre chose
```

Pointeurs *const* et fonctions membres *const*

Au Chapitre 6, vous avez appris à définir des fonctions constantes à l'aide du mot-clé `const`. Si une fonction `const` tente de modifier les données de l'objet, le compilateur indique une erreur.

Si vous déclarez un pointeur vers un objet constant, seules des méthodes constantes peuvent être appelées sur ce pointeur, comme le montre le Listing 8.10.

Listing 8.10 : Utilisation de pointeurs sur des objets constants

```
 1:  // Listing 8.10 - Pointeurs et méthodes const
 2:
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  class Rectangle
 7:  {
 8:  public:
 9:     Rectangle();
10:     ~Rectangle();
11:     void SetLongueur(int longueur) { saLongueur = longueur; }
12:     int GetLongueur() const { return saLongueur; }
13:     void SetLargeur(int largeur) { saLargeur = largeur; }
14:     int GetLargeur() const { return saLargeur; }
15:
16:  private:
17:     int saLongueur;
18:     int saLargeur;
19:  };
20:
21:  Rectangle::Rectangle()
22:  {
23:     saLargeur = 5;
24:     saLongueur = 10;
25:  }
26:
27:  Rectangle::~Rectangle()
28:  {}
29:
30:  int main()
31:  {
32:   Rectangle* pRect =  new Rectangle;
33:   const Rectangle * pRectConst = new Rectangle;
34:   Rectangle * const pPtrConst = new Rectangle;
35:
36:   cout << "Largeur de pRect : " << pRect->GetLargeur()
37:        << " m" << endl;
38:   cout << "Largeur de pRectConst : "
38a:       << pRectConst->GetLargeur()
39:        << " m" << endl;
40:   cout << "Largeur de pPtrConst: " << pPtrConst->GetLargeur()
41:        << " m" << endl;
```

```
 42:
 43:   pRect->SetLargeur(10);
 44:   // pRectConst->SetLargeur(10);
 45:   pPtrConst->SetLargeur(10);
 46:
 47:   cout << "Largeur de pRect : " << pRect->GetLargeur()
 48:        << " m\n";
 49:   cout << "Largeur de pRectConst : "
49a:        << pRectConst->GetLargeur()
 50:        << " m\n";
 51:   cout << "Largeur de pPtrConst : "
51a:        << pPtrConst->GetLargeur()
 52:        << " m\n";
 53:   return 0;
 54:  }
```

Ce programme produit le résultat suivant :

```
Largeur de pRect : 5 m
Largeur de pRectConst : 5 m
Largeur de pPtrConst : 5 m
Largeur de pRect : 10 m
Largeur de pRectConst : 5 m
Largeur de pPtrConst : 10 m
```

La classe `Rectangle` est définie de la ligne 6 à la ligne 19. La déclaration de la méthode constante `GetLargeur()` figure à la ligne 14. La ligne 32 déclare un pointeur sur `Rectangle`, nommé `pRect`. La ligne 33 déclare un pointeur `pRectConst` vers un `Rectangle` constant alors que la ligne 34 déclare un pointeur constant `pPtrConst` vers un `Rectangle`.

Les valeurs de ces trois pointeurs s'affichent aux lignes 36 à 41.

À la ligne 43, le pointeur `pRect` est utilisé pour modifier la largeur de son rectangle (10 mètres). À la ligne 44, le pointeur `pRectConst` devrait faire de même, mais il n'a pas le droit d'appeler une fonction membre non constante car c'est un pointeur vers un `Rectangle` constant. N'étant pas une instruction valide, nous l'avons placée en commentaire.

À la ligne 45, `pPtrConst` appelle à son tour la fonction `SetLargeur()`. Comme c'est un pointeur constant sur un rectangle, il ne peut pas pointer sur autre chose mais le rectangle lui-même n'étant pas constant, les méthodes comme `GetLargeur()` et `SetLargeur()` peuvent être appelées.

Pointeurs *const this*

Lorsque vous déclarez un objet `const`, vous déclarez en fait que le pointeur `this` de cet objet est un pointeur sur un objet constant. Un pointeur `const this` ne peut donc être utilisé qu'avec des fonctions membres constantes.

Faire	*Ne pas faire*
● Protéger les objets passés par référence à l'aide du mot-clé `const` s'ils ne doivent pas être modifiés. ● Affecter 0 aux pointeurs plutôt que de les laisser non-initialisés ou pendants.	● Utiliser un pointeur dont on a libéré l'emplacement qu'il pointait. ● Appeler plusieurs fois `delete` sur le même pointeur.

Nous reviendrons sur les objets constants et les pointeurs constants dans le prochain chapitre, lorsque nous présenterons les références vers des objets constants.

Questions-réponses

Q Pourquoi les pointeurs jouent-ils un rôle si important ?

R Les pointeurs sont importants pour de nombreuses raisons. Ils permettent de stocker les adresses des objets et de les passer en paramètre par référence. Au Chapitre 14, vous verrez comment on les utilise pour mettre en œuvre le polymorphisme de classe. En outre, de nombreux systèmes d'exploitation et bibliothèques de classes créent des objets en votre nom et renvoient des pointeurs vers ces objets.

Q À quoi sert le tas ?

R Les objets stockés sur le tas ne disparaissent pas après le retour d'une fonction. La possibilité de stocker des objets sur le tas vous permet de décider lors de l'exécution du nombre d'objets dont vous avez besoin au lieu d'être obligé de les déclarer à l'avance. Ceci sera présenté plus en détail dans le chapitre suivant.

Q Pourquoi déclarer un objet constant si cela limite ce que je peux en faire ?

R Comme tous les programmeurs, vous souhaitez que le compilateur détecte les erreurs avant de créer le fichier exécutable. Une des erreurs les plus sournoises est le cas d'une fonction qui modifie un objet d'une manière peu détectable dans la fonction appelante. Pour protéger les objets qui ne doivent pas être mis à jour, il suffit de les déclarer à l'aide du mot-clé `const`.

Testez vos connaissances

1. Quel opérateur permet d'extraire l'adresse d'une variable ?
2. Quel opérateur permet d'extraire la valeur stockée à l'emplacement pointé par un pointeur ?
3. Qu'est-ce qu'un pointeur ?
4. Quelle est la différence entre l'adresse stockée dans un pointeur et la valeur à cette adresse ?
5. Quelle est la différence entre l'opérateur d'indirection et l'opérateur *adresse de* ?
6. Quelle est la différence entre ces deux déclarations ?

```
const int * ptrUn;
int * const ptrDeux;
```

Exercices

1. Que signifient ces déclarations ?
   a. `int * pUn;`
   b. `int vDeux;`
   c. `int * pTrois = &vDeux;`
2. La variable `votreAge` est de type `unsigned short`. Déclarez un pointeur permettant de la manipuler.
3. Affectez la valeur 50 à la variable `votreAge` en utilisant le pointeur déclaré précédemment.
4. Écrivez un petit programme dans lequel vous allez déclarer un entier et un pointeur vers un entier. Affectez tout d'abord l'adresse de l'entier au pointeur, puis utilisez le pointeur pour affecter une valeur à la variable entière.
5. CHERCHEZ L'ERREUR : où l'erreur se cache-t-elle dans ce programme ?

```
#include <iostream>
using namespace std;
int main()
{
     int *pInt;
     *pInt = 9;
     cout << "Valeur à pInt: " << *pInt;
     return 0;
}
```

6. CHERCHEZ L'ERREUR : où l'erreur se cache-t-elle dans ce programme ?

```
#include <iostream>
using namespace std;
int main()
{
    int Variable = 5;
    cout << "Variable : " << Variable << endl;
    int *pVar = &Variable;
    pVar = 9;
    cout << "Variable : " << *pVar << endl;
return 0;
      }
```



9

Références

Au sommaire de ce chapitre

- Nature et rôle des références
- Différences entre les références et les pointeurs
- Création et utilisation des références
- Limites d'utilisation des références
- Passage par référence de valeurs et d'objets à des fonctions et récupération des valeurs renvoyées

Dans le chapitre précédent, vous avez appris à gérer des objets sur le tas et à vous y référer de façon indirecte. Les références présentent quasiment toutes les fonctionnalités des pointeurs, mais avec une syntaxe beaucoup plus simple.

Qu'est-ce qu'une référence ?

Une référence est un *alias*. En d'autres termes, une référence est initialisée avec le nom d'un autre objet appelé *objet cible*. Dès lors, elle se comporte comme l'objet cible et toute opération effectuée sur la référence est répercutée sur celui-ci.

Pour créer une référence, il suffit d'indiquer le type de l'objet cible, suivi de l'opérateur référence (`&`) et du nom de la référence, puis le signe égal et, enfin, le nom de l'objet cible.

Une référence peut porter n'importe quel nom de variable autorisé en C++. Certains programmeurs utilisent une convention consistant à préfixer le nom des références par la lettre `"r"`, comme ici :

```
int &rReference = VariableEntiere;
```

Dans cette instruction, `rReference` est une référence à une variable de type `int`, `VariableEntiere`. Les références sont différentes des autres variables que vous pouvez déclarer car elles *doivent* être initialisées lors de leur déclaration. Si vous tentez de créer une variable référence sans lui fournir une valeur initiale, le compilateur produit une erreur. Le Listing 9.1 montre comment créer et utiliser des références.



L'opérateur référence et l'opérateur adresse de utilisent le même symbole (`&`). Bien qu'étroitement liés, ces opérateurs sont différents.

L'espace avant l'opérateur référence est obligatoire, mais celui entre l'opérateur et le nom de la variable est facultatif. Par conséquent ces deux instructions sont correctes :

```
int &rReference = VariableEntiere;  // syntaxe OK
int & rReference = VariableEntiere; // syntaxe OK
```

Listing 9.1 : Création et utilisation de références

```
 1:  //Listing 9.1 - Utilisation des références
 2:
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     using namespace std;
 8:     int  intUn;
 9:     int &rUneRef = intUn;
10:
11:     intUn = 5;
12:     cout << "intUn : " << intUn << endl;
13:     cout << "rUneRef : " << rUneRef << endl;
```

```
14:
15:     rUneRef = 7;
16:     cout << "intUn : " << intUn << endl;
17:     cout << "rUneRef : " << rUneRef << endl;
18:
19:     return 0;
20: }
```

Ce programme produit le résultat suivant :

```
intUn : 5
rUneRef: 5
intUn : 7
rUneRef : 7
```

La ligne 8 déclare la variable locale `intUn` de type entier. La ligne suivante déclare une référence à un entier, `rUneRef`, et l'initialise pour qu'elle désigne `intUn`. Comme on l'a déjà mentionné, si vous oubliez d'initialiser une référence, le compilateur renvoie une erreur et ne crée pas le fichier objet.

La ligne 11 affecte 5 à la variable `intUn`. Les valeurs contenues dans `intUe` et `rUneRef` sont ensuite affichées aux lignes 12 et 13. Bien entendu, elles sont égales.

La ligne 15 affecte 7 à `rUneRef`. Comme il s'agit d'un alias de la variable `intUn`, cette dernière reçoit également la valeur 7, comme le montre les affichages des lignes 16 et 17.

Utilisation de l'opérateur *adresse de* (*&*) sur des références

Vous savez maintenant que le symbole `&` permet à la fois d'obtenir l'adresse d'une variable et de déclarer une référence. Mais que se passe-t-il si vous demandez l'adresse d'une variable référence ? Le programme renverra l'adresse de l'objet cible, ce qui est logique puisque la référence est un alias de la cible (voir Listing 9.2).

Listing 9.2 : Extraction de l'adresse d'une référence

```
1:  //Listing 9.2 - Utilisation des références
2:
3:  #include <iostream>
4:
5:  int main()
6:  {
7:     using namespace std;
8:     int  intUn;
```

```
 9:     int &rUneRef = intUn;
10:
11:     intOne = 5;
12:     cout << "intUn : " << intUn << endl;
13:     cout << "rUneRef : " << rUneRef << endl;
14:
15:     cout << "&intUn : "  << &intUn << endl;
16:     cout << "&rUneRef : " << &rUneRef << endl;
17:
18:     return 0;
19: }
```

Ce programme produit le résultat suivant :

```
intUn : 5
rUneRef : 5
&intUn :  0x3500
&rUneRef : 0x3500
```



Les deux dernières lignes affichent des adresses mémoire qui risquent d'être particulières à votre ordinateur ou à une exécution du programme ; le résultat que vous obtiendrez sera sûrement différent.

`rUneRef` est à nouveau initialisée pour faire référence à la variable `intUn`. Les adresses des deux variables s'affichent : elles sont identiques (lignes 15 et 16).

C++ ne permet pas d'accéder directement à l'adresse d'une référence car cela n'aurait aucun intérêt, contrairement à l'adresse d'un pointeur ou d'une variable. Les références étant toujours initialisées lors de leur création, elles se comportent toujours comme un synonyme de leur cible, même si on leur applique l'opérateur *adresse de*.

Supposons que l'on dispose d'une classe `President` ; vous pourriez créer une instance de celle-ci avec l'instruction suivante :

```
President UnPrez;
```

Vous pouvez ensuite créer une référence de `President` et l'initialiser avec cet objet :

```
President &ChefNation = unPrez;
```

Un seul `President` existe. Les deux identificateurs désignent le même objet de la même classe. Toute action effectuée sur `ChefNation` se répercutera donc sur `UnPrez`.

Il faut bien faite la distinction entre le symbole `&` figurant à la ligne 9 du Listing 9.2 – qui déclare une référence `rUneRef` à une variable entière – et les mêmes symboles aux lignes 15 et 16 – qui renvoient les adresses de la variable entière `intUn` et de la référence

`rUneRef`. Le compilateur fait la distinction entre ces deux utilisations grâce à leur contexte d'utilisation.



L'utilisation d'une référence ne nécessite pas l'opérateur adresse de. Il suffit d'utiliser la référence comme vous utiliseriez la variable cible.

Tentative de réaffecter des références (*ne le faites pas !)*

Les variables référence ne peuvent pas être réaffectées. Même des programmeurs chevronnés peuvent se laisser piéger en croyant réaffecter une référence alors qu'ils réaffectent simplement l'objet cible, comme le montre le Listing 9.3. Les variables de référence sont toujours des alias de leur cible.

Listing 9.3 : Affectation à une référence

```
 1:  //Listing 9.3 - Réaffectation d'une référence
 2:
 3:  #include <iostream>
 4:
 5:  int main()
 6:  {
 7:     using namespace std;
 8:     int  intUn;
 9:     int &rUneRef = intUn;
10:
11:     intUn = 5;
12:     cout << "intUn :    " << intUn << endl;
13:     cout << "rUneRef :  " << rUneRef << endl;
14:     cout << "&intUn :   "  << &intUn << endl;
15:     cout << "&rUneRef : " << &rUneRef << endl;
16:
17:     int intDeux = 8;
18:     rUneRef = intDeux;  // différent de ce que vous croyez !
19:     cout << "\nintUn :   " << intUn << endl;
20:     cout << "intDeux :    " << intDeux << endl;
21:     cout << "rUneRef :  " << rUneRef << endl;
22:     cout << "&intUn:   "  << &intUn << endl;
23:     cout << "&intDeux :   "  << &intDeux << endl;
24:     cout << "&rUneRef : " << &rUneRef << endl;
25:     return 0;
26:  }
```

Ce programme produit le résultat suivant :

```
intUn :          5
rUneRef :        5
&intUn :         0012FEDC
&rUneRef :       0012FEDC

intUn :          8
intDeux :        8
rUneRef :        8
&intUn :         0012FEDC
&intDeux :       0012FEE0
&UneRef :        0012FEDC
```

Cette fois encore, une variable entière et une référence à un entier sont déclarées aux lignes 8 et 9. La ligne 11 affecte 5 à l'entier et les valeurs et leurs adresses sont affichées aux lignes 12 à 15.

La ligne 17 définit et initialise la variable `intDeux`. À la ligne suivante, le programmeur tente de réaffecter `rUneRef` pour qu'elle devienne un alias de cette nouvelle variable, mais ce n'est pas ce qui se passe en réalité : `rUneRef` continue d'être un alias de `intUn` et cette affectation est donc équivalente à l'instruction suivante :

```
intUne = intDeux;
```

Comme l'affichage des lignes 19 à 21 le prouve, `intUn`, `rUneRef` et `intDeux` sont désormais égales. L'affichage des adresses prouve également que `rUneRef` fait toujours référence à `intUn`, pas à `intDeux`.

Faire	*Ne pas faire*
● Utiliser les références pour créer des alias d'objets. ● Initialiser toutes les références.	● Tenter de réaffecter une référence. ● Confondre l'opérateur *adresse de* et l'opérateur *référence*.

Référencement des objets

Tous les objets, y compris ceux définis par l'utilisateur, peuvent être référencés. Une référence ne portant jamais sur une classe, mais sur l'un de ses objets, cette expression est interdite :

```
int & rRefInt = int;    // erreur
```

La référence `rRefInt` doit être initialisée avec un entier spécifique, comme ici :

```
int Valeur = 200;
int & rRefInt = Valeur;
```

De la même façon, il est interdit d'initialiser une référence avec la classe `Chat` :

```
Chat & rRefChat = Chat;   // erreur
```

Vous devez initialiser une référence avec un objet de la classe `Chat`, comme ici :

```
Chat Mistigri;
Chat & rRefChat = Mistigri;
```

La référence à un objet est utilisée comme l'objet lui-même. Ainsi, les données et les méthodes membres sont accessibles comme avec n'importe quel objet de la classe, à l'aide de l'opérateur point (`.`), comme le montre le Listing 9.4.

Listing 9.4 : Références aux objets

```
 1:  // Listing 9.4 - Références à des objets d'une classe
 2:
 3:  #include <iostream>
 4:
 5:  class SimpleChat
 6:  {
 7:  public:
 8:     SimpleChat (int age, int poids);
 9:     ~SimpleChat() {}
10:     int GetAge() { return sonAge; }
11:     int GetPoids() { return sonPoids; }
12:  private:
13:     int sonAge;
14:     int sonPoids;
15:  };
16:
17:  SimpleChat::SimpleChat(int age, int poids)
18:  {
19:     sonAge = age;
20:     sonPoids = weight;
21:  }
22:
23:  int main()
24:  {
25:     SimpleChat Frisky(5,4);
26:     SimpleChat & rChat = Frisky;
27:
```

```
28:     std::cout << "Frisky a ";
29:     std::cout << Frisky.GetAge() << " ans. " << std::endl;
30:     std::cout << "Frisky pèse ";
31:     std::cout << rChat.GetPoids() << " kilos." std::endl;
32:     return 0;
33:  }
```

Ce programme produit le résultat suivant :

```
Frisky a 5 ans
Frisky pèse 4 kilos.
```

La ligne 25 déclare `Frisky` comme un objet de la classe `SimpleChat`. La ligne suivante crée une référence `rChat` vers un `SimpleChat` et l'initialise avec l'objet `Frisky`. Les lignes 29 à 31 font appel aux méthodes accesseurs en utilisant d'abord l'objet lui-même, puis sa référence. On remarque ce ces deux accès utilisent la même syntaxe puisqu'ici aussi, une référence est un alias de l'objet réel.

Références

Les références agissent comme un alias de variable. Déclarer une référence consiste à en indiquer le type, suivi de l'opérateur référence (&) et du nom de la référence. Les références doivent être initialisées lors de leur création.

Exemple 1

```
int sonAge;
int &rAge = sonAge;
```

Exemple 2

```
Chat Frisky;
Chat &rRefChat = Frisky;
```

Pointeurs nuls et références nulles

Lorsqu'un pointeur n'a pas été initialisé ou lorsque l'emplacement qu'il pointe a été supprimé, il doit être initialisé à zéro. Ceci ne s'applique pas aux références, car elles doivent être initialisées sur ce qu'elles référencent lors de la déclaration.

Toutefois, C++ étant utilisable par des pilotes de périphériques, des systèmes embarqués et des systèmes en temps réel qui peuvent accéder directement au matériel, il faut pouvoir référencer des adresses spécifiques. La plupart des compilateurs permettent

donc d'initialiser une référence à zéro ou à une autre valeur numérique ; le programme ne se plantera que si vous tentez d'utiliser l'objet et que cette référence est invalide.

Il n'est cependant pas conseillé d'utiliser cette possibilité pour une programmation classique. En effet, si un tel programme est porté d'une machine à une autre, de mystérieuses erreurs pourraient apparaître si vous utilisez des références nulles.

Passage de paramètres par référence

Au Chapitre 5, nous avons vu que les fonctions présentent deux restrictions : les paramètres sont passés par valeur et l'instruction `return` ne peut renvoyer qu'une seule valeur.

Passer des valeurs par référence permet de contourner ces deux problèmes. En C++, vous pouvez passer des paramètres par référence de deux façons : en utilisant des pointeurs ou des références. Vous passez *par référence* en utilisant un pointeur ou vous passez *une référence* en utilisant une référence.

La syntaxe qui utilise un pointeur est différente de celle qui utilise une référence, mais l'effet est le même : c'est l'objet d'origine qui est mis à la disposition de la fonction, non une copie de celui-ci

Passer un objet par référence permet à la fonction de modifier l'objet référencé. Au Chapitre 5, vous avez appris que les paramètres des fonctions sont passés *via* la pile. Lorsqu'une valeur est passée par référence – avec un pointeur ou une référence –, seule l'adresse de l'objet original est placée sur la pile, pas l'objet lui-même. En réalité, sur certains ordinateurs, l'adresse est stockée dans un registre, ce qui évite d'utiliser la pile. Dans les deux cas, comme c'est une adresse qui est transmise, le compilateur peut identifier l'objet d'origine et les modifications auront lieu sur celui-ci, pas sur une copie.

Dans le Listing 5.5 du Chapitre 5, nous avions utilisé une fonction de permutation appelée `swap()` qui n'affectait pas les valeurs dans la fonction appelante. À titre de rappel, le Listing 9.5 reproduit cet exemple.

Listing 9.5 : Passage de paramètres par valeur

```
1:  //Listing 9.5 - Passage par valeur
2:  #include <iostream>
3:
4:  using namespace std;
5:  void swap(int x, int y);
6:
7:  int main()
8:  {
9:     int x = 5, y = 10;
```

```
10:
11:     cout << "Dans Main. Avant échange, x : " << x << " y : "
11a:         << y << endl;
12:     swap(x, y);
13:     cout << "Dans Main. Après échange, x : " << x << " y : "
13a:         << y << endl;
14:     return 0;
15:  }
16:
17:  void swap (int x, int y)
18:  {
19:     int temp;
20:
21:     cout << "Dans Swap. Avant échange, x : " << x << " y: "
21a:         << y << endl;
22:
23:     temp = x;
24:     x = y;
25:     y = temp;
26:
27:     cout << "Dans Swap. Après échange, x : " << x << " y : "
27a:         << y << endl;
28:  }
```

Ce programme produit le résultat suivant :

```
Dans Main. Avant échange, x : 5 y : 10
Dans Swap. Avant échange, x : 5 y : 10
Dans Swap. Après échange, x : 10 y : 5
Dans Main. Après échange, x : 5 y : 10
```

Ce programme initialise deux variables dans la fonction `main()`, puis les passe à la fonction `swap()` qui semble les intervertir. De retour dans la fonction principale, ces valeurs n'ont pas été modifiées !

Le problème, ici, est que `x` et `y` ont été passées par valeur et que la fonction en a donc créé des copies locales. Ces copies ont donc été modifiées et ont été ensuite supprimées à la fin de la fonction, en même temps que ses variables locales. Il est préférable de les passer par référence, afin de modifier les valeurs sources des variables, et non des copies locales.

Pour ce faire, vous avez deux solutions en C++ : soit vous faites en sorte que les paramètres de la fonction `swap()` soient des pointeurs sur les valeurs d'origine, soit vous passez en paramètre des références aux valeurs originales.

Utilisation de pointeurs avec la fonction *swap()*

Si vous passez des pointeurs en paramètre, vous transmettez en fait les adresses des objets, ce qui fait que la fonction peut modifier les valeurs situées à ces adresses. `swap()` doit donc être déclarée pour attendre deux pointeurs d'entiers. En déréférençant ces pointeurs, la fonction pourra alors accéder aux véritables valeurs de `x` et `y` et les échanger. C'est ce que fait le Listing 9.6

Listing 9.6 : Passage par référence à l'aide de pointeurs

```
 1:  //Listing 9.6 Passage par référence
 2:  #include <iostream>
 3:
 4:  using namespace std;
 5:  void swap(int *x, int *y);
 6:
 7:  int main()
 8:  {
 9:     int x = 5, y = 10;
10:
11:     cout << "Dans Main. Avant échange, x : " << x << " y : "
11a:         << y << endl;
12:     swap(&x,&y);
13:     cout << "Dans Main. Après échange, x : " << x << " y : "
13a:         << y << endl;
14:     return 0;
15:  }
16:
17:  void swap (int *px, int *py)
18:  {
19:     int temp;
20:
21:     cout << "Dans Swap. Avant échange, *px : " << *px <<
22:        " *py : " << *py << endl;
23:
24:     temp = *px;
25:     *px = *py;
26:     *py = temp;
27:
28:     cout << "Dans Swap. Après échange, *px : " << *px <<
29:        " *py : " << *py << endl;
30:
31:  }
```

Ce programme produit le résultat suivant :

```
Dans Main. Avant échange, x : 5 y : 10
Dans Swap. Avant échange, *px : 5 *py : 10
Dans Swap. Apres échange, *px : 10 *py : 5
Dans Main. Apres échange, x : 10 y : 5
```

Ça marche ! À la ligne 5, le prototype de la fonction `swap()` indique que les paramètres sont désormais des pointeurs vers des entiers au lieu de variables entières. Lors de l'appel la fonction, en ligne 12, on passe en paramètre les adresses de `x` et de `y` : on peut voir que ce sont des adresses qui sont passées car l'on utilise l'opérateur *adresse de* (`&`).

La ligne 19 déclare la variable locale `temp` dans la fonction `swap()`. Il ne s'agit pas d'un pointeur, mais d'une variable qui contiendra la valeur de `*px` (c'est-à-dire la valeur de l'emplacement pointé par `px`, qui correspond à celle de `x` dans la fonction appelante). Lorsque la fonction se termine, cette variable locale disparait de la pile.

La valeur à l'adresse `px` est affectée à `temp` à la ligne 24, l'emplacement pointé par `px` recevant ensuite la valeur située à l'adresse `py` (ligne 25). La valeur stockée dans la variable `temp` est ensuite copiée dans l'emplacement pointé par `py`.

Les valeurs de la fonction appelante dont on a transmis les adresses à la fonction `swap()` sont ainsi interverties.

Utililisation de références avec la fonction *swap()*

Bien que programme précédent fasse ce que l'on attend, la syntaxe de la fonction `swap()` est un peu lourde. Les opérations répétées de déréférencement des pointeurs sont difficiles à relire et sujettes aux erreurs – si vous oubliez de déréférencer le pointeur, le compilateur vous permettra quand même de lui affecter un entier et c'est l'utilisateur de votre fonction qui subira l'erreur. En outre, devoir explicitement passer en paramètre les adresses des variables de la fonction appelante lors de l'appel expose aux utilisateurs le fonctionnement interne de la fonction `swap()`.

Un langage orienté objet comme C++ se doit de masquer les détails de fonctionnement interne d'une fonction. L'utilisation des fonction doit être totalement transparente pour les utilisateurs. L'utilisation de paramètres pointeurs reporte donc la charge du passage par référence sur la fonction appelante alors que cela devrait être géré par la fonction appelée. Le Listing 9.7 propose une autre écriture de `swap()`, qui utilise cette fois-ci des paramètres références.

Listing 9.7 : Fonction *swap()* réécrite avec des références

```
 1:  //Listing 9.7 Passage par référence
 2:  // à l'aide de références !
 3:  #include <iostream>
 4:
 5:  using namespace std;
 6:  void swap(int &x, int &y);
 7:
 8:  int main()
 9:  {
10:     int x = 5, y = 10;
11:
12:     cout << "Dans Main. Avant échange, x : " << x << " y : "
13:        << y << endl;
14:
15:     swap(x,y);
16:
17:     cout << "Dans Main. Après échange, x : " << x << " y  "
18:        << y << endl;
19:
20:     return 0;
21:  }
22:
23:  void swap (int &rx, int &ry)
24:  {
25:     int temp;
26:
27:     cout << "Dans Swap. Avant échange, rx : " << rx
28:          << " ry : " << ry << endl;
29:
30:     temp = rx;
31:     rx = ry;
32:     ry = temp;
33:
34:
35:     cout << "Dans Swap. Après échange, rx : " << rx
36:          << " ry : " << ry << endl;
37:
38:  }
```

Ce programme produit le résultat suivant :

```
Dans Main. Avant échange, x : 5 y : 10
Dans Swap. Avant échange, rx : 5 ry : 10
Dans Swap. Après échange, rx : 10 ry : 5
Dans Main. Après échange, x : 10, y : 5
```

Comme dans l'exemple précédent, le programme commence par déclarer deux variables (ligne 10) et affiche leurs valeurs (ligne 12). La ligne 15 appelle la fonction `swap()` en lui passant en paramètre les variables `x` et `y` (et non leurs adresses).

Le programme va alors directement à la ligne 23, où ces paramètres sont identifiés comme des références. Les valeurs de ces variables sont affichées à la ligne 27 et vous pouvez constater qu'il n'est pas nécessaire d'utiliser des opérateurs spéciaux : ces variables sont des alias des variables d'origine et peuvent donc être utilisées comme telles.

Une fois interverties (lignes 30 à 32), on affiche à nouveau les valeurs puis le programme revient en la ligne 17 pour afficher les variables de la fonction `main()`. Les paramètres de `swap()` étant des références, les opérations réalisées dans la fonction appelée se sont répercutées au variables initiales de la fonction appelante.

Comme on le voit dans ce listing, les références permettent donc d'utiliser des variables normales et de disposer de toute la puissance du passage par référence avec les pointeurs !

En-têtes et prototypes de fonctions

Dans les deux listings ci-dessus, les programmes ont utilisé successivement des pointeurs et des références. L'utilisation des références est plus aisée et rend le code plus facile à comprendre mais comment la fonction appelante peut-elle savoir si les paramètres ont été passés par valeur ou par référence ? En tant que client (ou utilisateur) de `swap()`, le programmeur doit s'assurer que cette fonction modifiera bien ses paramètres.

Pour le savoir, il peut examiner les paramètres déclarés dans le prototype de la fonction. Ce dernier se trouve généralement dans un fichier en-tête, accompagné d'autres prototypes. Grâce à ces informations, le programmeur peut donc savoir que ses valeurs seront passées par référence à la fonction `swap()` et seront donc correctement échangées. Le prototype `swap()` apparaît à la ligne 6 du Listing 9.7 : on peut constater que les deux paramètres sont passés comme des références.

Si `swap()` avait été une fonction membre d'une classe, la déclaration de classe également disponible dans le fichier en-tête aurait fourni ces informations.

En C++, les clients des classes et des fonctions peuvent se fier aux fichiers en-têtes pour connaître tout ce qu'ils ont besoin de savoir. Ce fichier agit alors comme une interface vers la classe ou la fonction, dont l'implémentation est cachée à l'utilisateur. Celui-ci peut alors se consacrer à son problème et utiliser les classes ou les fonctions sans se soucier des détails de leur fonctionnement.

Lors de la construction du pont de Brooklyn, le colonel John Roebling supervisait totalement les travaux, de la fabrication des matériaux (câbles, briques et ciment) au respect des plans. De nos jours, les ingénieurs utilisent mieux leur temps en ne s'occupant que de la

mise en œuvre des matériaux, sans s'occuper de savoir comment ils ont été fabriqués du moment qu'ils correspondent au cahier des charges.

La démarche est la même en C++. Les programmeurs utilisent des classes et des fonctions éprouvées sans se préoccuper de leur fonctionnement internes. Ces "composants logiciels" peuvent être assemblés pour produire un programme, exactement comme les câbles, les briques et le ciment sont assemblés pour fabriquer des immeubles et des ponts.

Comme un ingénieur des ponts et chaussées se reporte à la documentation technique pour connaître la capacité, le volume et le débit d'un tuyau d'évacuation, par exemple, le développeur examine la déclaration d'une fonction ou d'une classe pour connaître les services qu'elle fournit, les paramètres qu'elle attend et les résultats qu'elle produit.

Renvoi de plusieurs valeurs

Comme nous l'avons vu plus haut, une fonction ne peut renvoyer qu'une seule valeur à la fois. Comment faire pour en renvoyer deux ? La première solution consiste à passer deux objets par référence à la fonction. Le passage par référence autorisant une fonction à modifier les objets d'origine, elle peut donc renvoyer deux éléments d'information. Cette pratique permet de se passer de la valeur renvoyée par la fonction, qui peut par exemple servir à signaler une erreur.

Là encore, on peut utiliser des références ou des pointeurs. Dans le Listing 9.8, la fonction renvoie trois valeurs, la valeur de renvoi habituelle et deux paramètres pointeurs.

Listing 9.8 : Renvoi de valeurs à l'aide de pointeurs

```
 1:  //Listing 9.8 - Renvoi de plusieurs valeurs par une fonction
 2:
 3:  #include <iostream>
 4:
 5:  using namespace std;
 6:  short Facteur(int n, int* pCarre, int* pCube);
 7:
 8:  int main()
 9:  {
10:     int nombre, carre, cube;
11:     short erreur;
12:
13:     cout << "Entrez un nombre entre 0 et 20 : ";
14:     cin >> nombre;
15:
16:     erreur = Facteur(nombre, &carre, &cube);
17:
```

```
18:     if (!erreur)
19:     {
20:        cout << "Nombre : " << nombre << endl;
21:        cout << "Carré : " << carre << endl;
22:        cout << "Cube : "  << cube   << endl;
23:     }
24:     else
25:        cout << "Erreur !" << endl;
26:     return 0;
27:  }
28:
29:  short Facteur(int n, int *pCarre, int *pCube)
30:  {
31:     short Valeur = 0;
32:     if (n > 20)
33:        Valeur = 1;
34:     else
35:     {
36:        *pCarre = n*n;
37:        *pCube = n*n*n;
38:        Valeur = 0;
39:     }
40:     return Valeur;
41:  }
```

Ce programme produit le résultat suivant :

```
Entrez un nombre entre 0 et 20 : 3
Valeur : 3
Carré : 9
Cube : 27
```

La ligne 10 définit les variables de type entier court, `nombre`, `carre`, et `cube`. Puis `nombre` reçoit la valeur saisie par l'utilisateur à la ligne 14. La ligne 16 transmet cette valeur et les adresses de `carre` et `cube` à la fonction `Facteur()`.

La ligne 32 examine le premier paramètre, qui a été passé par valeur. S'il est supérieur à 20 (valeur maximale gérée par la fonction), la valeur renvoyée (`Valeur`) correspondra à une erreur, 0 indiquant que tout s'est bien passé. `Valeur` est renvoyée à la fonction appelante à la ligne 40.

Les valeurs correspondant au carré et au cube du nombre saisi ne sont pas renvoyées de manière traditionnelle mais en modifiant les emplacements pointés par les deux paramètres pointeurs passés à la fonction.

Les valeurs de renvoi sont donc affectées aux pointeurs (lignes 36 et 37). Ces valeurs sont affectées aux variables initiales au moyen du mécanisme d'indirection, que l'on peut

repérer par l'utilisation de l'opérateur de déréférencement (*) sur les noms de pointeurs. À la ligne 38, `Valeur` reçoit la valeur 0 signifiant une fin normale du traitement et elle est renvoyée à l'appelant à la ligne 40.



Le passage par référence ou par pointeur permettant un accès non contrôlé aux attributs d'objets et aux méthodes, il est conseillé de ne passer que le minimum requis pour que la fonction réalise sa tâche. C'est un gage de sécurité et de lisibilité.

Renvoi de valeurs par référence

Bien que le Listing 9.8 fonctionne correctement, il est conseillé de privilégier les références aux pointeurs car la maintenance et la relecture du code seront plus faciles. Le Listing 9.9 correspond au Listing 9.8 mais utilise des références à la place des pointeurs.

Il inclut également une autre amélioration puisqu'il utilise un `enum` pour que la valeur de retour soit plus simple à comprendre. Au lieu de renvoyer 0 ou 1, cet `enum` permet à la fonction de renvoyer `SUCCES` ou `ECHEC`.

Listing 9.9 : Utilisation de références

```
 1:  //Listing 9.9
 2:  // Renvoi de plusieurs valeurs par une fonction
 3:  // à l'aide de références
 4:  #include <iostream>
 5:
 6:  using namespace std;
 7:
 8:  enum CODE_ERREUR { SUCCES, ECHEC };
 9:
10:  CODE_ERREUR Facteur(int, int&, int&);
11:
12:  int main()
13:  {
14:     int nombre, carre, cube;
15:     CODE_ERREUR resultat;
16:
17:     cout << "Entrez un nombre entre 0 et 20 : ";
18:     cin >> nombre;
19:
20:     resultat = Facteur(nombre, carre, cube);
21:
22:     if (resultat == SUCCES)
23:     {
24:        cout << "Nombre : " << nombre << endl;
```

```
25:        cout << "Carré : " << carre << endl;
26:        cout << "Cube : "  << cube << endl;
27:     }
28:     else
29:        cout << "Erreur !" << endl;
30:     return 0;
31:  }
32:
33:  CODE_ERREUR Facteur(int n, int &rCarre, &rCube)
34:  {
35:     if (n > 20)
36:        return ECHEC;   // simple code d'erreur
37:     else
38:     {
39:        rCarre = n*n;
40:        rCube = n*n*n;
41:        return SUCCES;
42:     }
43:  }
```

Ce programme produit le résultat suivant :

```
Entrez un nombre entre 0 et 20 : 3
Valeur : 3
Carré : 9
Cube : 27
```

Le Listing 9.9 est identique au précédent, à deux exceptions près : l'énumération `CODE_ERREUR` permet de renvoyer un code d'erreur bien plus explicite (lignes 36 et 41) et la gestion des erreurs est plus complète (ligne 22).

Notez que la fonction `Facteur()` est désormais déclarée pour attendre des références aux variables `carre` et `cube`, et non des pointeurs. Ceci facilite à la fois la manipulation de ces paramètres et la relecture du code.

Efficacité du passage de paramètres par référence

Le passage par valeur d'un objet à une fonction implique une copie de cet objet. Lorsque la fonction appelée renvoie un objet par valeur, une autre copie a lieu.

Au Chapitre 5, nous avons vu que ces objets sont stockés sur la pile. Ces opérations prennent du temps et consomme de la mémoire. Ces coûts soient négligeables pour les petits objets comme les entiers.

En revanche, ils ne le sont plus du tout pour les objets plus volumineux, créés par l'utilisateur. Dans la pile, la taille d'un tel objet correspond en effet à la somme de toutes les variables membres qui le composent. Ces variables sont parfois elles-mêmes des objets définis par l'utilisateur. Passer en paramètre une structure aussi massive en la copiant sur la pile peut avoir un impact important sur les performances et la consommation mémoire du programme.

Un autre coût s'ajoute encore. Avec les classes que vous créez, chacune de ces copies temporaires est créée par un appel à un constructeur spécial, appelé *constructeur de copie*, qui est automatiquement appelé par le compilateur . Au Chapitre 10, vous apprendrez à les utiliser et à créer vos propres constructeurs de copies. Pour l'instant, il vous suffit de savoir que le constructeur de copie est appelé à chaque fois qu'une copie temporaire de l'objet est placé sur la pile.

Lorsque cet objet temporaire est détruit à la fin de l'exécution de la fonction, son destructeur est appelé. Si la fonction renvoie un objet par valeur, il faut également faire une copie de cet objet puis le détruire.

Avec des objets volumineux, ces appels de constructeurs et de destructeurs peuvent avoir un impact sur la vitesse d'exécution du programme et sur son utilisation de la mémoire. Pour illustrer ce phénomène, le Listing 9.10 présente une version épurée de la classe `SimpleChat` – les véritables objets manipulés par les programmes sont plus gros et plus coûteux, mais cet exemple suffit à montrer la fréquence des appels du constructeur de copie et du destructeur.

Listing 9.10 : Passage d'objets par référence

```
 1:  //Listing 9.10 - Passage de pointeurs sur des objets
 2:
 3:  #include <iostream>
 4:
 5:  using namespace std;
 6:  class SimpleChat
 7:  {
 8:  public:
 9:     SimpleChat ();                  // constructeur
10:     SimpleChat(SimpleChat&);        // constructeur de copie
11:     ~SimpleChat();                  // destructeur
12:  };
13:
14:  SimpleChat::SimpleChat()
15:  {
16:     cout << "Constructeur de SimpleChat..." << endl;
17:  }
18:
```

```
19:  SimpleChat::SimpleChat(SimpleChat&)
20:  {
21:     cout << "Constructeur de copie de SimpleChat..." << endl;
22:  }
23:
24:  SimpleChat::~SimpleChat()
25:  {
26:     cout << "Destructeur de SimpleChat..." << endl;
27:  }
28:
29:  SimpleChat FonctionUn (SimpleChat leChat);
30:  SimpleChat* FonctionDeux (SimpleChat *leChat);
31:
32:  int main()
33:  {
34:     cout << "Création d'un objet Chat..." << endl;
35:     SimpleChat Frisky;
36:     cout << "Appel de FonctionUn..." << endl;
37:     FonctionUn(Frisky);
38:     cout << "Appel de FonctionDeux..." << endl;
39:     FonctionDeux(&Frisky);
40:     return 0;
41:  }
42:
43:  // FonctionUn, passage par valeur
44:  SimpleChat FonctionUn(SimpleChat leChat)
45:  {
46:     cout << "Retour de FonctionUn..." << endl;
47:     return leChat;
48:  }
49:
50:  // FonctionDeux, passage par référence
51:  SimpleChat* FonctionDeux(SimpleChat  *leChat)
52:  {
53:     cout << "Retour de FonctionDeux..." << endl;
54:     return leChat;
55:  }
```

Ce programme produit le résultat suivant :

```
Création de l'objet Chat...
Constructeur de SimpleChat...
Appel de FonctionUn...
Constructeur de copie de SimpleChat...
Retour de FonctionUn...
```

```
Constructeur de copie de SimpleChat...
Destructeur de SimpleChat...
Destructeur de SimpleChat...
Appel de FonctionDeux...
Retour de FonctionDeux...
Destructeur de SimpleChat...
```

Le Listing 9.10 crée l'objet `SimpleChat`, puis appelle deux fonctions. La première reçoit `leChat` par valeur, puis le renvoie par valeur. La seconde reçoit un pointeur vers l'objet, et non l'objet lui-même, puis renvoie un pointeur vers l'objet.

Les lignes 6 à 12 déclarer une classe `SimpleChat` minimale. Le constructeur, le constructeur de copie et le destructeur se contentent de produire un message pour vous informer du déroulement du programme.

À la ligne 34, la fonction `main()` affiche le premier message qui apparaît dans le résultat. La ligne suivante instancie un objet `SimpleChat`, ce qui provoque l'appel du constructeur et donc l'affichage de la deuxième ligne du résultat

La ligne 36 signale l'appel de `FonctionUn()`. Cette dernière reçoit l'objet `SimpleChat` par valeur, ce qui provoque la création d'une copie locale sur la pile et donc l'appel du constructeur de copie, qui affiche la quatrième ligne du résultat.

Le programme va ensuite directement à la ligne 46, dans la fonction appelée qui affiche la cinquième ligne du résultat. La fonction principale reprend la main et récupère l'objet `SimpleChat` renvoyé par valeur, ce qui a pour effet de créer une autre copie de l'objet et donc d'appeler le constructeur de copie qui affiche la sixième ligne du résultat.

La valeur renvoyée par `FonctionUn()` n'étant affectée à aucun objet, le destructeur supprime l'objet temporaire et affiche la septième ligne. `FonctionUn()` s'étant terminée, sa copie locale devient hors de portée et est détruite par le destructeur qui affiche la huitième ligne.

La fonction principale reprend le contrôle et appelle la fonction `FonctionDeux()` en passant le paramètre par référence, ce qui ne produit pas de copie de l'objet sur la pile. `FonctionDeux()` affiche le message qui apparaît à la dixième ligne du résultat et renvoie l'objet `SimpleChat` par référence, ce qui évite là aussi d'appeler le constructeur et le destructeur.

Enfin, le programme se termine et `Frisky` sort de sa portée, ce qui cause un dernier appel au destructeur, qui affiche la dernière ligne du résultat.

La fonction `FonctionUn()` ayant reçu le paramètre `SimpleChat` par valeur a donc provoqué deux appels du constructeur de copie et deux appels du destructeur, alors que `FonctionDeux()` n'en a produit aucun.

Passer un pointeur *const*

Le passage d'un pointeur à `FonctionDeux()` a beau être plus efficace, il n'en demeure pas moins dangereux. Cette fonction n'est, en effet, pas censée modifier l'objet qui lui est passé, or elle reçoit son adresse ce qui expose franchement l'objet original à des modifications et brise la protection offerte par le passage par valeur.

Le passage par valeur revient à donner à un musée une photographie de la pièce maîtresse de votre collection : si un vandale l'abime, l'original ne sera pas touché. Le passage par référence, au contraire, consiste à donner votre adresse au musée et à inviter les visiteurs à venir chez vous voir cette œuvre.

La solution consiste à passer un pointeur sur une constante `SimpleChat`, ce qui aura pour effet d'interdire l'appel de méthodes modificatrices sur cet objet et donc de le protéger contre les modifications.

Le passage d'une référence constante permet aux visiteurs de contempler votre peinture, mais leur interdit de l'altérer de quelque façon que ce soit. Le Listing 9.11 donne un exemple d'utilisation d'un paramètre pointeur vers une constante.

Listing 9.11 : Passage de pointeur sur un objet constant

```
1:  //Listing 9.11 - Passage de pointeurs d'objets
2:
3:  #include <iostream>
4:
5:  using namespace std;
6:  class SimpleChat
7:  {
8:  public:
9:     SimpleChat();
10:    SimpleChat(SimpleChat&);
11:    ~SimpleChat();
12:
13:    int GetAge() const { return sonAge; }
14:    void SetAge(int age) { sonAge = age; }
15:
16: private:
17:    int sonAge;
18: };
19:
20: SimpleChat::SimpleChat()
21: {
22:    cout << "Constructeur de SimpleChat..." << endl;
23:    wAge = 2;
24: }
```

```
25:
26:  SimpleChat::SimpleChat(SimpleChat&)
27:  {
28:     cout << "Constructeur de copie de SimpleChat..." << endl;
29:  }
30:
31:  SimpleChat::~SimpleChat()
32:  {
33:     cout << "Destructeur de SimpleChat..." << endl;
34:  }
35:
36:  const SimpleChat * const FonctionDeux
37:     (const SimpleChat * const leChat);
38:
39:  int main()
40:  {
41:     cout << "Création d'un objet Chat..." << endl;
42:     SimpleChat Frisky;
43:     cout << "Frisky a " ;
44:     cout << Frisky.GetAge();
45:     cout << " ans" << endl;
46:     int age = 5;
47:     Frisky.SetAge(age);
48:     cout << "Frisky a " ;
49:     cout << Frisky.GetAge();
50:     cout << " ans" << endl;
51:     cout << "Appel de FonctionDeux..." << endl;
52:     FonctionDeux(&Frisky);
53:     cout << "Frisky a " ;
54:     cout << Frisky.GetAge();
55:     cout << " ans" << endl;
56:     return 0;
57:  }
58:
59:  // FonctionDeux attend un pointeur const
60:  const SimpleChat * const FunctionTwo
61:     (const SimpleChat * const leChat)
62:  {
63:     cout << "Fin de FonctionDeux..." << endl;
64:     cout << "Frisky a maintenant " << leChat->GetAge();
65:     cout << " ans " << endl;
66:     // leChat->SetAge(8);   const !
67:     return leChat;
68:  }
```

Ce programme produit le résultat suivant :

```
Création d'un objet Chat...
Constructeur de SimpleChat...
Frisky a 2 ans
Frisky a 5 ans
Appel de FonctionDeux...
Fin de FonctionDeux...
Frisky a maintenant 5 ans
Frisky a 5 ans
Destructeur de SimpleChat...
```

`SimpleChat` contient deux fonctions d'accès : l'une est constante (`GetAge()`, à la ligne 13), l'autre, non constante (`SetAge()`, à la ligne 14). La variable membre `sonAge` est définie à la ligne 17.

Les appels respectifs au constructeur, au constructeur de copie et au destructeur sont signalés par des messages. On constate que le constructeur de copie n'est jamais appelé puisque l'objet est passé par référence et qu'aucune copie n'est donc réalisée. La ligne 42 crée un objet dont l'âge par défaut est affiché par la ligne suivante.

La ligne 47 modifie la variable `sonAge` en appelant la méthode d'accès `SetAge()` et la ligne suivante affiche cette nouvelle valeur. `FonctionDeux()` a été légèrement modifiée : son paramètre, comme sa valeur de retour, est désormais un pointeur constant sur un objet constant (ligne 36).

Le paramètre et la valeur de renvoi étant toujours passés par référence, aucune copie n'est réalisée et le constructeur de copie n'est donc pas sollicité. Cependant, l'objet pointé dans la fonction `FonctionDeux()` est désormais constant et ne peut donc pas appeler la fonction `SetAge()` qui, elle, ne l'est pas. Si la ligne 66 n'avait pas été mise en commentaire, le programme n'aurait pas pu pourrait pas être compilé.

L'objet créé dans la fonction `main()` n'étant pas constant, `Frisky` peut appeler `SetAge()`. L'adresse de cet objet non constant est passée à la fonction `FonctionDeux()` mais, comme cette fonction déclare son paramètre comme étant un pointeur constant vers un objet constant, l'objet est traité comme s'il était constant !

Une autre solution : le passage de références

Dans le Listing 9.11, le programme passe les paramètres par référence, ce qui évite de créer des copies locales et donc d'appeler le constructeur de copie et le destructeur. Il utilise pour cela des pointeurs constants sur des objets constants afin d'empêcher la fonction de modifier l'objet. Toutefois, le passage de pointeurs exige une notation assez lourde.

Comme on sait que ces objets ne seront jamais nuls, il serait plus simple de travailler sur des références, comme le montre le Listing 9.12.

Listing 9.12 : Passage de références à des objets

```
 1:  //Listing 9.12 - Passage de références à des objets
 2:
 3:  #include <iostream>
 4:
 5:  using namespace std;
 6:  class SimpleChat
 7:  {
 8:  public:
 9:     SimpleChat();
10:     SimpleChat(SimpleChat&);
11:     ~SimpleChat();
12:
13:     int GetAge() const { return sonAge; }
14:     void SetAge(int age) { sonAge = age; }
15:
16:  private:
17:     int sonAge;
18:  };
19:
20:  SimpleChat::SimpleChat()
21:  {
22:     cout << "Constructeur de SimpleChat..." << endl;
23:     wAge = 2;
24:  }
25:
26:  SimpleChat::SimpleChat(SimpleChat&)
27:  {
28:     cout << "Constructeur de copie de SimpleChat..." << endl;
29:  }
30:
31:  SimpleChat::~SimpleChat()
32:  {
33:     cout << "Destructeur de SimpleChat..." << endl;
34:  }
35:
36:  const SimpleChat & FonctionDeux(const SimpleChat & leChat);
37:
38:  int main()
39:  {
40:     cout << "Création d'un objet Chat..." << endl;
41:     SimpleChat Frisky;
42:     cout << "Frisky a " << Frisky.GetAge() << " ans" << endl;
```

```
43:     int age = 5;
44:     Frisky.DefAge(age);
45:     cout << "Frisky a " << Frisky.GetAge() << " ans" << endl;
46:     cout << "Appel de FonctionDeux...\n";
47:     FonctionDeux(Frisky);
48:     cout << "Frisky a " << Frisky.GetAge() << " ans" << endl;
49:     return 0;
50: }
51:
52: // FonctionDeux, attend une référence à un objet const
53: const SimpleChat & FonctionDeux(const SimpleChat & leChat)
54: {
55:     cout << "Fin de FonctionDeux..." << endl;
56:     cout << "Frisky a maintenant " << leChat.GetAge();
57:     cout << " ans " << endl;
58:     // leChat.SetAge(8);   const !
59:     return leChat;
60: }
```

Ce programme produit le résultat suivant :

```
Création d'un objet Chat...
Constructeur de SimpleChat...
Frisky a 2 ans
Frisky a 5 ans
Appel de FunctionDeux
Fin de FunctionDeux...
Frisky a maintenant 5 ans
Frisky a 5 ans
Destructeur de SimpleChat...
```

Le résultat est identique à celui du Listing 9.11. La fonction `FonctionDeux()` attend et renvoie désormais une référence à un objet constant. Vous pouvez constater une nouvelle fois qu'il est plus simple de travailler avec des références qu'avec des pointeurs et que l'on en tire les mêmes avantages en termes de performances, tout en conservant la sécurité offerte par `const`.

Références `const`

En général, les programmeurs C++ ne font pas la différence entre une "référence constante à un objet de la classe" et une "référence à un objet constant de la classe". Il est vrai qu'une référence est toujours constante, puisqu'elle ne peut jamais être réaffectée à un autre objet. Lorsque le mot-clé `const` s'applique à une référence, il concerne donc l'objet référencé.

Choisir entre références et pointeurs

Les développeurs expérimentés préfèrent souvent les références aux pointeurs parce qu'elles sont plus faciles à utiliser, plus lisibles et qu'elles permettent de mieux masquer les informations, comme on l'a vu dans le Listing 9.12.

Néanmoins, une référence ne peut pas être réaffectée. Si vous devez désigner d'abord un objet, puis un autre, vous devez donc utiliser un pointeur. En outre, une référence ne pouvant jamais être nulle, vous ne pouvez pas utiliser une référence si l'objet désigné risque d'être nul ; vous devez utiliser un pointeur.

Supposons que vous vouliez réserver un emplacement mémoire pour un objet. Si le mot-clé `new` est incapable d'allouer cet emplacement sur le tas, il renvoie un pointeur nul. Une référence ne pouvant jamais être nulle, vous ne devez pas initialiser une référence avec cet emplacement tant que vous n'avez pas vérifié qu'il n'est pas nul. L'exemple suivant montre comment procéder :

```
int *pInt = new int;
if (pInt != NULL)
  int &rInt = *pInt;
```

Dans cet exemple, le pointeur `pInt` est déclaré et initialisé avec l'adresse mémoire renvoyée par l'opérateur `new`. L'adresse contenue dans ce pointeur est ensuite testée. Si elle n'est pas nulle, `pInt` est déréférencé. Le résultat de l'opération produit un objet entier et `rInt` est initialisé pour y faire référence. `rInt` est donc devenu un alias de l'entier renvoyé par `new`.

Faire	*Ne pas faire*
● Passer des paramètres par référence le plus souvent possible. ● Protéger les références et les pointeurs à l'aide du mot-clé `const`.	● Utiliser des pointeurs lorsqu'il est possible de se servir de références. ● Tenter de réaffecter une référence à une autre variable. C'est impossible.

Mélanger les références et les pointeurs dans une liste de paramètres

Il est tout à fait possible de mélanger des objets passés par valeur, des pointeurs et des références dans la même liste de paramètres. Voici un exemple :

```
Chat * Fonction(Utilisateur &Proprio, Objet *unObjet, int age);
```

Cette déclaration indique que `Fonction` attend trois paramètres. Le premier est une référence à un objet `Utilisateur`, le deuxième est un pointeur sur un objet `Objet` et le troisième est un entier passé par valeur. Elle renvoie un pointeur vers un objet `Chat`.

L'emplacement de l'opérateur d'indirection (`*`) et de l'opérateur de référence (`&`) dans ces déclarations de paramètres est sujet à controverse parmi les programmeurs. Les trois écritures suivantes sont tout à fait correctes lorsque vous voulez déclarer une référence :

```
1:  Chat&  rMistigri;
2:  Chat & rMistigri;
3:  Chat  &rMistigri;
```

Les espaces n'étant pas pris en compte au niveau syntaxique, vous pouvez insérer autant de tabulations, d'espaces et de lignes que vous le souhaitez.

La question est de savoir quelle est l'expression la plus adaptée ? Voici des arguments pour ces trois syntaxes :

Dans le premier cas, le paramètre est une variable appelée `rMistigri`, dont le type peut être considéré comme une "référence à un objet `Chat`". L'opérateur `&` doit donc être associé au type.

Le contre-argument est que le type est `Chat`. L'opérateur `&` fait partie du "déclarateur", qui comprend le nom de la variable et l'esperluette. Lorsqu'il est collé au nom du type, l'opérateur est source d'erreur dans l'expression suivante :

```
Chat& rMistigri, rFrisky;
```

A priori, `rMistigri` et `rFrisky` laissent à penser qu'il s'agit de références à des objets de la classe `Chat`. En fait, le premier élément est ici une référence à un objet `Chat`, alors que `rFrisky` (malgré son nom) est une variable classique de la classe `Chat`. Cette ligne devrait donc être écrite de la façon suivante :

```
Chat &rMistigri, rFrisky;
```

La réponse à cette objection est qu'il ne faudrait jamais mélanger de cette façon les déclarations de références et de variables. Voici les déclarations équivalentes correctes :

```
Chat& rMistigri;
Chat  Frisky;
```

La plupart des programmeurs optent généralement pour une solution intermédiaire, en plaçant l'opérateur entre les éléments de la liste (voir la deuxième proposition).

Bien entendu, ce qui s'applique à l'opérateur référence (`&`) concerne également l'opérateur d'indirection (`*`). Chacun utilise les opérateurs à sa façon, l'essentiel étant de rester homogène et cohérent dans les programmes.



Les deux conventions suivantes sont fréquemment adoptées pour la déclaration de pointeurs et de références :

- *L'esperluette et l'astérisque occupent une position médiane, en étant précédés et suivis d'un espace.*
- *Ne jamais déclarer des références, des pointeurs et des variables sur la même ligne.*

Renvoi de références d'objet hors de portée

Les programmeurs C++ qui ont pris l'habitude de passer des paramètres par référence ont tendance à abuser de cette pratique. Une référence correspond à l'alias d'un objet existant. Lorsque vous passez ou récupérez une référence dans une fonction, posez-vous la question suivante : "Quel est l'objet que je référence et existera-t-il toujours à chaque fois qu'il est utilisé ?".

Le Listing 9.13 illustre le danger du renvoi d'une référence à un objet qui n'existe plus.

Listing 9.13 : Renvoi d'une référence à un objet inexistant

```
 1:  // Listing 9.13
 2:  // Renvoi d'une référence à un objet
 3:  // qui n'existe plus
 4:
 5:  #include <iostream>
 6:
 7:  class SimpleChat
 8:  {
 9:  public:
10:     SimpleChat (int age, int poids);
11:     ~SimpleChat() {}
12:     int GetAge() { return sonAge; }
13:     int GetPoids() { return sonPoids; }
14:  private:
15:     int sonAge;
16:     int sonPoids;
17:  };
18:
19:  SimpleChat::SimpleChat(int age, int poids)
20:  {
21:     sonAge = age;
22:     sonPoids = poids;
23:  }
24:
25:  SimpleChat &UneFonction();
```

```
26:
27:  int main()
28:  {
29:     SimpleChat &rChat = UneFonction();
30:     int age = rChat.GetAge();
31:     std::cout << "rChat a " << age << " ans !" << std::endl;
32:     return 0;
33:  }
34:
35:  SimpleChat &UneFonction()
36:  {
37:     SimpleChat Frisky(5, 9);
38:     return Frisky;
39:  }
```



Ce programme ne pourra pas être compilé avec un compilateur Borland, alors qu'il le sera avec un compilateur Microsoft ou avec le compilateur GNU. Cette façon de procéder n'est toutefois pas recommandée.

La classe `SimpleChat` est déclarée de la ligne 7 à la ligne 17. Une référence à un objet `SimpleChat` est ensuite initialisée avec le résultat de l'appel à la fonction `UneFonction()` (ligne 29), qui est déclarée comme devant renvoyer une référence à cette classe, comme l'indique la ligne 25.

Le corps de la fonction `UneFonction()` [lignes 35-39] déclare un objet local de type `SimpleChat`, et initialise son âge et son poids. Cet objet est ensuite renvoyé par référence à la fonction appelante. Certains compilateurs sont capables de détecter cette erreur et donc d'empêcher la création du fichier exécutable, alors que d'autres compilent le fichier source (en produisant éventuellement un message d'avertissement) et créent ainsi un programme qui produira des résultats imprévisibles.

Lorsque la fonction `UneFonction()` se termine, l'objet local `Frisky` est détruit. La référence renvoyée par cette fonction est donc un alias d'un objet qui n'existe plus, ce qui est incorrect.

Renvoi d'une référence à un objet dans le tas

On pourrait être tenté de résoudre le problème précédent en faisant en sorte que la fonction `UneFonction()` crée `Frisky` sur le tas. De cette façon, `Frisky` continuerait d'exister après l'exécution de la fonction.

Le problème de cette approche est alors ce qu'il advient de la mémoire allouée à l'objet `Frisky` lorsque l'on n'en a plus besoin. Le Listing 9.14 illustre ce problème.

Listing 9.14 : Fuites mémoire

```
// Listing 9.14 - Résoudre les problèmes de fuites mémoire

#include <iostream>

class SimpleChat
{
public:
   SimpleChat (int age, int poids);
   ~SimpleChat() {}
   int GetAge() { return sonAge; }
   int GetPoids() { return sonPoids; }

private:
   int sonAge;
   int sonPoids;
};

SimpleChat::SimpleChat(int age, int weight)
{
   sonAge = age;
   sonPoids = poids;
}

SimpleChat & UneFonction();

int main()
{
   SimpleChat & rChat = UneFonction();
   int age = rChat.GetAge();
   std::cout << "rChat a " << age << " ans !" << std::endl;
   std::cout << "&rChat : " << &rChat << std::endl;
   // Comment se débarrasser de cette mémoire ?
   SimpleChat * pChat = &rChat;
   delete pChat;
   // rChat fait maintenant référence à ??
   return 0;
}

SimpleChat &UneFonction()
{
   SimpleChat * pFrisky = new SimpleChat(5, 9);
   std::cout << "pFrisky : " << pFrisky << std::endl;
   return *pFrisky;
}
```

Ce programme produit le résultat suivant :

```
pFrisky : 0x00431C60
rChat a 5 ans !
&rChat : 0x00431C60
```



Une fois compilé, ce programme semble fonctionner correctement. Il s'agit en fait d'une véritable bombe à retardement.

La fonction `UneFonction()` [lignes 39 à 44] ne renvoie plus de référence à une variable locale. La mémoire est allouée sur le tas et affectée à un pointeur à la ligne 41. L'adresse correspondante est affichée, puis le pointeur est déréférencé, ce qui permet de renvoyer l'objet `SimpleChat` par référence.

La ligne 28 affecte le résultat de cette fonction à une référence de `SimpleChat` et cet objet est utilisé pour obtenir l'âge du chat, qui est affiché à la ligne 30.

La référence déclarée dans la fonction `main()` désigne l'objet placé sur le tas par `UneFonction()`. Pour le prouver, on applique l'opérateur *adresse de* à `rChat`. Les deux adresses qui s'affichent renvoient la même valeur : l'objet cité par référence et son adresse dans le tas sont identiques.

Pour l'instant, tout va bien, mais comment libérer cet espace mémoire ? Souvenez-vous qu'il est interdit d'appeler `delete` sur une référence. Une solution astucieuse consiste à créer un autre pointeur, pour l'initialiser ensuite avec l'adresse obtenue par `rChat`. La mémoire est libérée et on évite ainsi une fuite mémoire. Mais un autre problème subsiste : sur quel objet la référence `rChat` porte-t-elle après la ligne 34 ? Comme vous l'avez vu plus haut, une référence doit toujours être l'alias d'un objet existant non nul. Si l'objet est nul ou n'existe pas (comme c'est le cas ici), le programme n'est pas valide.



N'utilisez jamais de références à des objets nuls. En effet, les programmes peuvent parfois être compilés, mais ils produiront des erreurs inattendues lors de leur exécution.

Il existe trois solutions à ce problème. La première consiste à déclarer un objet `SimpleChat` à la ligne 28, puis à le renvoyer par valeur à la fonction appelante. La deuxième consiste à déclarer un objet `SimpleChat` sur le tas dans `UneFonction()` mais faire en sorte que celle-ci renvoie un pointeur vers cet emplacement : la fonction appelante pourra alors libérer la zone pointée par ce pointeur lorsqu'elle n'en aura plus besoin. Enfin, la dernière solution (de loin la meilleure) consiste à déclarer l'objet dans la fonction appelante, puis à le passer par référence à `UneFonction()`.

Où le pointeur est-il passé ?

Lorsqu'un programme alloue de l'espace mémoire sur le tas, il faut impérativement conserver un pointeur vers cet emplacement. Dans le cas contraire, la mémoire ne peut plus être libérée ce que entraîne une fuite mémoire.

Durant le déroulement du programme, ce pointeur passe d'une fonction à une autre. En règle générale, la valeur placée dans le bloc de mémoire est passée à l'aide de références et c'est la fonction qui a créé cet emplacement qui le supprime également. Mais il ne s'agit que d'une règle générale qui n'est pas gravée dans le marbre.

Toutefois, il est dangereux de supprimer une zone mémoire créé dans une fonction à partir d'une autre fonction. En effet, le pointeur risque de ne pas être supprimé ou de l'être deux fois, ce qui produit des erreurs d'exécution dans le programme. Il est plus sûr de construire vos fonctions de sorte qu'elles libèrent les espaces qu'elles ont elles-mêmes alloués.

Si vous écrivez une fonction qui doit allouer de la mémoire puis la repasser à la fonction appelante, vous devriez sans doute modifier votre interface pour faire en sorte que ce soit la fonction appelante qui alloue la mémoire, puis la passe par référence à votre fonction. Cette approche déplace toute la gestion de la mémoire de votre programme vers la fonction qui est préparée pour la supprimer.

Faire	*Ne pas faire*
● Passer les paramètres par valeur si nécessaire. ● Renvoyer le résultat d'une fonction par valeur si nécessaire.	● Renvoyer un élément par référence si sa portée est locale. ● Perdre la trace du moment et de l'endroit où la mémoire est allouée car vous risquez ainsi de ne pas savoir si elle est libérée.

Questions-réponses

Q À quoi bon utiliser des références alors que les pointeurs sont aussi performants ?

R Les références sont plus faciles à employer et à comprendre. L'indirection est cachée, et il est inutile de déréférencer la variable à chaque accès.

Q Si les références sont plus faciles à mettre en œuvre, pourquoi utilise-t-on encore des pointeurs ?

R Les références ne peuvent jamais être nulles et ne peuvent pas être réaffectées. Les pointeurs offrent plus de souplesse, mais sont un peu plus difficiles à utiliser.

Q Pourquoi le résultat d'une fonction doit-il être renvoyé par valeur ?

R Si l'objet renvoyé est local à la fonction, vous devez le renvoyer par valeur sous peine de renvoyer une référence vers un objet inexistant.

Q Puisque le renvoi par référence est dangereux, pourquoi ne pas toujours renvoyer un résultat par valeur ?

R Le renvoi par référence est plus efficace. Il économise la mémoire et le programme s'exécute plus vite.

Testez vos connaissances

1. Quelle est la différence entre une référence et un pointeur ?
2. Quand faut-il préférer les pointeurs aux références ?
3. Que renvoie `new` lorsqu'il n'y a plus assez de mémoire pour créer un nouvel objet ?
4. Qu'est-ce qu'une référence constante ?
5. Quelle est la différence entre le passage d'objets par référence et le passage d'une référence ?
6. Lors de la déclaration d'une référence, que faut-il écrire ?

   a. `int&   maRef = monInt;`

   b. `int  &  maRef = monInt;`

   c. `int   &maRef = monInt;`

Exercices

1. Dans un programme, déclarez une variable entière, une référence à cet objet et un pointeur vers un entier. Utilisez le pointeur et la référence pour manipuler la valeur de l'entier.
2. Écrivez un programme déclarant un pointeur constant sur un entier constant. Initialisez le pointeur avec l'adresse d'une variable entière `Var1`. Affectez 6 à `Var1` et utilisez le pointeur pour affecter 7 à `Var1`. Créez ensuite une autre variable entière `Var2` et réaffectez le pour qu'il pointe sur `Var2`. Ne compilez pas encore cet exercice.
3. Compilez maintenant le programme de l'Exercice 2. Produit-il des erreurs ? Quels sont les messages d'avertissement ?
4. Écrivez un programme qui crée un pointeur perdu.

5. Corrigez ce programme.
6. Écrivez un programme provoquant une fuite mémoire.
7. Corrigez ce programme.
8. CHERCHEZ L'ERREUR : où se cache l'erreur dans ce programme ?

```
#include <iostream>
using namespace std;
class CHAT
{
   public:
      CHAT(int age) { sonAge = age; }
      ~CHAT(){}
      int GetAge() const { return sonAge;}
   private:
      int sonAge;
};

CHAT & CreerChat(int age);
int main()
{
   int age = 7;
   CHAT Felix = CreerChat(age);
   cout << "Felix a " << Felix.GetAge()
        << " ans" << endl;
return 0;
}

CHAT & CreerChat(int age)
{
   CHAT * pChat = new CHAT(age);
   return *pChat;
}
```

9. Corrigez le programme de l'exercice précédent.



10

Fonctions avancées

Au sommaire de ce chapitre

- La surcharge de fonctions membres
- La surcharge des opérateurs
- Les fonctions gérant des classes avec des variables membres allouées dynamiquement

Au Chapitre 5, vous avez appris à utiliser des fonctions. Vous connaissez à présent le fonctionnement des pointeurs et des références. Dans ce chapitre, nous allons approfondir notre connaissance des fonctions.

Fonctions membres surchargées

Au Chapitre 5, nous avons vu comment implémenter le polymorphisme de fonctions, ou surcharge de fonctions, qui consiste à créer au moins deux fonctions de même nom mais avec des paramètres différents. Cette technique concerne également les fonctions membres des classes (ou méthodes).

Dans le Listing 10.1, la classe `Rectangle` contient deux fonctions `DessinerForme()`. L'une ne prend aucun paramètre et trace le rectangle en fonction de ses valeurs courantes alors que l'autre prend deux valeurs (largeur et hauteur) et les utilise pour tracer le rectangle sans tenir compte de ses valeurs actuelles.

Listing 10.1 : Surcharge de fonctions membres

```
 1:  //Listing 10.1 Surcharge de fonctions membres de la classe
 2:  #include <iostream>
 3:
 4:  // Déclaration de la classe Rectangle
 5:  class Rectangle
 6:  {
 7:  public:
 8:     // constructeurs
 9:     Rectangle(int largeur, int hauteur);
10:     ~Rectangle(){}
11:
12:     // fonction DessinerForme surchargée
13:     void DessinerForme() const;
14:     void DessinerForme(int uneLargeur, int uneHauteur) const;
15:
16:  private:
17:     int saLargeur;
18:     int saHauteur;
19:  };
20:
21:  // implémentation du constructeur
22:  Rectangle::Rectangle(int largeur, int hauteur)
23:  {
24:     saLargeur = largeur;
25:     saHauteur = hauteur;
26:  }
27:
28:
29:  // fonction DessinerForme surchargée sans paramètres
30:  // Dessine en fonction des valeurs membres courantes
30a:  // de l'objet
31:  void Rectangle::DessinerForme() const
32:  {
33:     DessinerForme(saLargeur, saHauteur);
34:  }
35:
36:
```

```
37:  // fonction DessinerForme surchargée - prend deux valeurs
38:  // Dessine en fonction des paramètres passés
39:  void Rectangle::DessinerForme(int largeur, int hauteur) const
40:  {
41:     for (int i = 0; i < hauteur; i++)
42:     {
43:        for (int j = 0; j < largeur; j++)
44:        {
45:           std::cout << "*";
46:        }
47:     std::cout << std::endl;
48:     }
49:  }
50:
51:  // Programme principal de démonstration
51a: // des fonctions surchargées
52:  int main()
53:  {
54:     // initialise un rectangle de 30x5
55:     Rectangle unRect(30,5 );
56:     std::cout << "DessinerForme() :" << std::endl;
57:     unRect.DessinerForme();
58:     std::cout << "\nDessinerForme(40, 2) :" << std::endl;
59:     unRect.DessinerForme(40,2);
60:     return 0;
61:  }
```

Ce programme produit le résultat suivant :

```
DessinerForme() :
******************************
******************************
******************************
******************************
******************************

DessinerForme(40, 2) :
****************************************************************
****************************************************************
```

La partie la plus importante du programme réside dans la surcharge de la fonction `DessinerForme()` (lignes 13 et 14). La mise en œuvre des méthodes surchargées s'effectue de la ligne 31 à la ligne 49. Vous remarquerez que la version de `DessinerForme()` qui ne prend aucun paramètre se contente d'appeler l'autre version en lui passant les variables membres courantes. Essayez toujours d'éviter de dupliquer le code dans deux fonctions car cela complique les modifications ultérieures et multiplie les risques d'erreur.

Le programme commence réellement à la ligne 52 et se poursuit jusqu'à la ligne 61. Il crée un objet `Rectangle`, appelle la première version de la méthode `DessinerForme()` (sans paramètre) puis la seconde (en lui passant deux paramètres entiers courts non signés).

Le compilateur détermine la méthode qui doit être appelée en utilisant le nombre et le type de paramètres passés à l'appel. On pourrait imaginer une troisième méthode `DessinerForme()` qui prendrait en paramètre une dimension et une énumération qui indiquerait s'il s'agit de la hauteur ou de la largeur.

Utilisation de valeurs par défaut

Tout comme les fonctions globales, les méthodes d'une classe peuvent inclure une ou plusieurs valeurs par défaut, comme le montre le Listing 10.2.

Listing 10.2 : Utilisation de valeurs par défaut

```
 1:  //Listing 10.2 Valeurs par défaut dans les fonctions membres
 2:  #include <iostream>
 3:
 4:  using namespace std;
 5:
 6:  // Déclaration de la classe Rectangle
 7:  class Rectangle
 8:  {
 9:  public:
10:     // constructeurs
11:     Rectangle(int largeur, int hauteur);
12:     ~Rectangle(){}
13:     void DessinerForme(int uneLargeur, int uneHauteur,
14:                        bool ValsActuelles = false) const;
15:
16:  private:
17:     int saLargeur;
18:     int saHauteur;
19:  };
20:
21:  // Implémentation du constructeur
22:  Rectangle::Rectangle(int largeur, int hauteur):
23:    saLargeur(largeur),       // initialisations
24:    saHauteur(hauteur)
25:  {}                      // le corps est vide
26:
27:
```

```
// valeurs par défaut utilisées pour le 3ème paramètre
void Rectangle::DessinerForme(
   int largeur,
   int hauteur,
   bool ValeursActuelles
  ) const
{
   int printLargeur;
   int printHauteur;

   if (ValeursActuelles == true)
   {
      // utiliser les valeurs actuelles
      printLargeur = saLargeur;
      printHauteur = saHauteur;
   }
   else
   {
      printLargeur = largeur;    // utiliser les paramètres
      printHauteur = hauteur;
   }


   for (int i = 0; i < printHauteur; i++)
   {
      for (int j = 0; j < printLargeur; j++)
      {
         cout << "*";
      }
      cout << endl;
   }
}

// Programme principal pour la démonstration
// des fonctions surchargées
int main()
{
   // Initialise un rectangle de 30x5
   Rectangle unRect(30, 5);
   cout << "DessinerForme(0, 0, true)..." << endl;
   unRect.DessinerForme(0, 0, true);
   cout <<"DessinerForme(40, 2)..." << endl;
   unRect.DessinerForme(40, 2);
   return 0;
}
```

Ce programme produit le résultat suivant :

```
DessinerForme(0, 0, true)...
******************************
******************************
******************************
******************************
******************************

DessinerForme(40, 2)...
************************************************************
************************************************************
```

Dans ce listing, la fonction surchargée `DessinerForme()` est remplacée par une unique fonction avec trois paramètres par défaut (ligne 14). Les deux premiers, `uneLargeur` et `uneHauteur`, sont de type entier, alors que le troisième, de type `bool` (`ValsActuelles`), est égal à `false` par défaut.

Cette fonction quelque peu complexe est implémentée à partir de la ligne 29. Vous savez que les espaces n'ont pas d'importance en C++ ; l'en-tête de la fonction se trouve donc aux lignes 29 à 33.

Dans la méthode, le troisième paramètre (`ValeursActuelles`) est évalué à la ligne 38. S'il vaut `true`, les variables locales `printLargeur` et `printHauteur` reçoivent respectivement les valeurs courantes des membres `saLargeur` et `saHauteur`.

Dans le cas contraire, soit que le paramètre ait été initialisé par défaut à `false` ou défini par l'utilisateur, les deux premiers paramètres servent à initialiser les variables `printLargeur` et `printHauteur`.

Si `ValeursActuelles` vaut `true`, les valeurs des deux autres paramètres ne sont pas prises en compte.

Choisir entre des valeurs par défaut et des fonctions surchargées

Les deux premiers programmes de ce chapitre réalisent les mêmes opérations. Pourtant, les fonctions surchargées du Listing 10.1 sont plus faciles à utiliser et à comprendre. En outre, il est plus simple d'étendre la surcharge en créant, par exemple, une troisième fonction si l'utilisateur souhaite entrer uniquement la hauteur ou la largeur du rectangle mais pas les deux. En revanche, une valeur par défaut deviendrait rapidement très complexe à mesure que l'on ajouterait des variantes.

Comment choisir entre la surcharge de fonction et l'utilisation de valeurs par défaut ? Voici une règle très simple :

Préférez la surcharge de fonction dans les cas suivants :

- Il n'existe pas de valeurs par défaut significatives.
- Vous devez utiliser des algorithmes différents.
- Vous devez pouvoir gérer des types différents dans la liste des paramètres.

Le constructeur par défaut

Le rôle d'un constructeur est de définir un objet ; celui d'un constructeur `Rectangle`, par exemple, est de construire un objet rectangle valide. Avant le lancement du constructeur, il n'existe pas d'objet rectangle mais uniquement un espace en mémoire. Après son exécution, il existe un objet prêt à l'emploi. C'est un avantage essentiel de la programmation orientée objet : le programme appelant n'a rien à faire pour s'assurer que l'objet démarre dans un état cohérent.

Comme nous l'avons vu au Chapitre 6, si vous ne définissez pas explicitement de constructeur pour une classe, le compilateur crée pour vous un constructeur par défaut, qui n'attend aucun paramètre et qui ne fait rien. Bien entendu, vous avez la possibilité de déclarer votre propre constructeur par défaut, dépourvu de paramètres, mais permettant de "mettre en place" l'objet comme vous le souhaitez

Ce constructeur est appelé constructeur "par défaut" bien qu'il soit défini par l'utilisateur – ce terme désigne en fait tout constructeur qui n'attend pas de paramètre.



Si vous définissez un ou plusieurs constructeurs dans une classe, le compilateur ne produira pas de constructeur par défaut pour celle-ci. Si vous en avez besoin d'un, vous devez donc le créer vous-même.

Surcharge des constructeurs

Les constructeurs, comme n'importe quelle fonction membre, peuvent être surchargés, ce qui leur ajoute de la puissance et de la souplesse.

Par exemple, les objets rectangles pourraient avoir deux constructeurs. L'un tracerait un rectangle à partir de la longueur et de la hauteur entrées par l'utilisateur alors que l'autre ne prendrait aucune valeur et définirait un rectangle par défaut. Le Listing 10.3 implémente ces deux constructeurs.

Listing 10.3 : Surcharge d'un constructeur

```
 1:  // Listing 10.3 - surcharge de constructeurs
 2:
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  class Rectangle
 7:  {
 8:  public:
 9:     Rectangle();
10:     Rectangle(int largeur, int longueur);
11:     ~Rectangle() {}
12:     int GetLargeur() const { return saLargeur; }
13:     int GetLongueur() const { return saLongueur; }
14:  private:
15:     int saLargeur;
16:     int saLongueur;
17:  };
18:
19:  Rectangle::Rectangle()
20:  {
21:     saLargeur = 5;
22:     saLongueur = 10;
23:  }
24:
25:  Rectangle::Rectangle (int largeur, int longueur)
26:  {
27:     saLargeur = largeur;
28:     saLongueur = longueur;
29:  }
30:
31:  int main()
32:  {
33:     Rectangle Rect1;
34:     cout << "Largeur de Rect1 : "
34a:         << Rect1.GetLargeur() << endl;
35:     cout << "Longueur de Rect1 : "
35a:         << Rect1.GetLongueur() << endl;
36:
37:     int uneLargeur, uneLongueur;
38:     cout << "Entrez la largeur : ";
39:     cin >> uneLargeur;
40:     cout << "\nEntrez la longueur : ";
41:     cin >> uneLongueur;
42:
43:     Rectangle Rect2(uneLargeur, uneLongueur);
```

```
 44:     cout << "\nLargeur de Rect2 : "
44a:          << Rect2.GetLargeur() << endl;
 45:     cout << "Longueur de Rect2 : "
45a:          << Rect2.GetLongueur() << endl;
 46:     return 0;
 47: }
```

Ce programme produit le résultat suivant :

```
Largeur de Rect1 : 5
Longueur de Rect1 : 10
Entrez la largeur  : 20

Entrez la longueur : 50

Largeur de Rect2 : 20
Longueur de Rect2 : 50
```

La classe `Rectangle` est déclarée de la ligne 6 à la ligne 17. Le programme comprend deux constructeurs : le constructeur "par défaut" (défini à la ligne 9) et un constructeur (ligne 10) prenant en paramètre deux valeurs entières.

La ligne 33 crée un rectangle à l'aide du constructeur par défaut et les lignes 34 et 35 affichent ses dimensions. L'utilisateur est ensuite invité à entrer une largeur et une longueur (lignes 38 à 41) qui sont transmises au constructeur qui attend deux paramètres (ligne 43). Enfin, la largeur et la hauteur de ce rectangle sont affichées.

Comme avec toute fonction surchargée, le compilateur appelle le constructeur qui convient en fonction du nombre et du type des paramètres.

Initialisation des objets

Jusqu'à présent, vous avez défini les variables membres des objets dans le corps du constructeur. Les constructeurs, cependant, sont invoqués en deux étapes : la phase d'initialisation et l'exécution de leurs corps.

La plupart des variables peuvent être définies durant l'une ou l'autre de ces étapes, soit lors de la phase initialisation, soit par affectation de valeurs dans le corps du constructeur. Il est plus propre et généralement plus lisible d'initialiser les variables membres lors de la phase d'initialisation. Voici un exemple d'initialisation des variables membre d'un `Chat` :

```
Chat():          // nom et paramètres du constructeur
  sonAge(5),     // liste d'initialisation
  sonPoids(8)
{ }              // corps du constructeur
```

La parenthèse fermante de la liste des paramètres du constructeur doit être suivie du caractère deux-points. Placez ensuite le nom de la variable membre, suivi de deux parenthèses entre lesquelles vous pouvez indiquer une ou plusieurs expressions qui initialisent la variable membre. S'il y plusieurs initialisations, séparez-les par une virgule.

Dans le Listing 10.4, nous avons choisi d'initialiser les variables membres dans la partie d'initialisation du constructeur au lieu de leur affecter des valeurs dans le corps, comme nous l'avions fait dans le Listing 10.3.

Listing 10.4 : Extrait de code initialisant des variables membres

```
 1: Listing 10.4 - Initialisation de variables membres
 2:  Rectangle::Rectangle():
 3:      saLargeur(5),
 4:      saLongueur(10)
 5:  {
 6:  }
 7:
 8:  Rectangle::Rectangle (int largeur, int longueur):
 9:      saLargeur(largeur),
10:      saLongueur(longueur)
11:  {
12:  }
```

Le Listing 10.4 n'étant qu'un extrait de code, il n'affiche donc rien. La ligne 2 débute le constructeur par défaut. Après l'en-tête standard, on a ajouté un caractère deux-points, suivi des définitions des valeurs par défaut `5` et `10` pour `saLargeur` et `saLongueur` aux lignes 3 et 4.

La ligne 8 débute la définition du deuxième constructeur. Dans cette version surchargée, on attend deux paramètres qui servent à initialiser les membres de la classe aux lignes 9 et 10.

Certaines variables, comme les références ou les constantes, doivent être initialisées, mais pas affectées. Le corps d'un constructeur contient souvent d'autres affectations ou des instructions diverses, mais il est toujours préférable d'utiliser le plus possible la phase d'initialisation.

Le constructeur de copie

Outre un constructeur par défaut et un destructeur, le compilateur produit également un constructeur de copie par défaut. Le constructeur de copie est appelé chaque fois qu'une copie d'objet est créée sur la pile ou sur le tas.

Comme nous l'avons vu au Chapitre 9, le passage d'un objet par valeur en paramètre à une fonction ou comme valeur de retour d'une fonction provoque la création d'une copie temporaire de cet objet. S'il s'agit d'une instance d'une classe définie par l'utilisateur, c'est le constructeur de copie de cette classe qui est appelé (voir le Listing 9.6 dans le chapitre précédent).

Un constructeur de copie ne prend toujours qu'un seul paramètre : une référence à un objet de la même classe. De préférence, ce sera une référence constante afin d'éviter toute modification malencontreuse de l'objet qui a été passé. Exemple :

```
Chat(const Chat & leChat);
```

Le constructeur de `Chat` attend ici une référence constante à un objet existant de la classe `Chat`. Le rôle de ce constructeur est de créer une copie de l'objet `leChat`.

Le constructeur de copie par défaut se contente de copier chaque variable membre de l'objet passé en paramètre dans les variables membres du nouvel objet. C'est ce que l'on appelle une *copie de surface* : bien qu'elle convienne à la plupart des variables membres, elle pose de sérieux problèmes lorsque ces membres sont des pointeurs sur des objets alloués sur le tas.

Une *copie de surface* duplique les valeurs exactes des variables membres d'un objet dans un autre objet. Les pointeurs de ces deux objets finissent donc par pointer sur le même espace mémoire. Une *copie en profondeur*, au contraire, copie les valeurs allouées sur le tas dans une nouvel emplacement, différent du premier.

Si la classe `Chat` contient une variable membre appelée `sonAge`, qui pointe sur un entier alloué sur le tas, le constructeur de copie par défaut copiera le contenu de la variable `sonAge` de l'objet passé en paramètre dans la variable membre `sonAge` de l'objet nouvellement créé. Ces deux objets pointeront alors sur la même adresse mémoire (voir Figure 10.1).

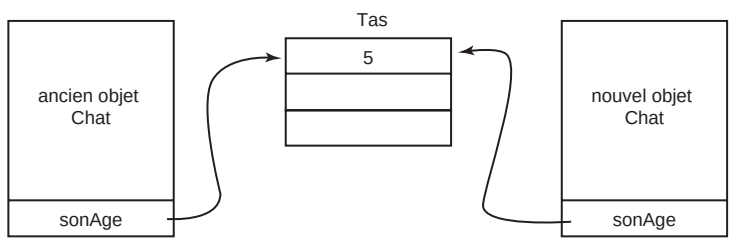


Figure 10.1
Utilisation du constructeur de copie par défaut.

Cette situation mènera au désastre si l'un des deux objets `Chat` sort de sa portée. Si le destructeur du `Chat` initial libère son emplacement alors que l'objet dupliqué pointe

toujours sur cette zone, on a créé un pointeur perdu et le programme court un danger mortel. La Figure 10.2 illustre ce problème.

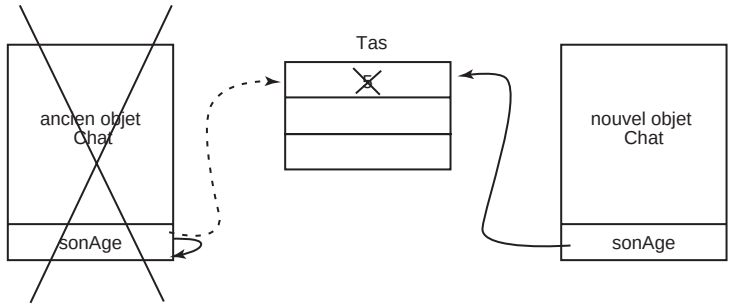


Figure 10.2
Création d'un pointeur perdu.

La solution consiste donc à créer votre propre constructeur de copie et à allouer l'espace mémoire en fonction de vos besoins. Une fois que vous avez alloué une nouvelle zone mémoire, vous pouvez copier les anciennes valeurs dans ce nouvel emplacement : vous avez réalisé une copie en profondeur, comme on le montre dans le Listing 10.5.

Listing 10.5 : Constructeurs de copie

```
// Listing 10.5 - constructeurs de copie

#include <iostream>
using namespace std;

class Chat
{
public:
   Chat();                // constructeur par défaut
   Chat (const Chat &);   // constructeur de copie
   ~Chat();               // destructeur
   int GetAge()   const   { return *sonAge; }
   int GetPoids()   const   { return *sonPoids; }
   void SetAge(int age)   { *sonAge = age; }

private:
   int *sonAge;
   int *sonPoids;
};

Chat::Chat()
{
   sonAge = new int;
   sonPoids = new int;
```

```
25:     *sonAge = 5;
26:     *sonPoids = 9;
27:  }
28:
29:  Chat::Chat(const Chat & rhs)
30:  {
31:     sonAge = new int;
32:     sonPoids = new int;
33:     *sonAge = rhs.GetAge();        // accès public
34:     *sonPoids = *(rhs.sonPoids);   // accès privé
35:  }
36:
37:  Chat::~Chat()
38:  {
39:     delete sonAge;
40:     sonAge = 0;
41:     delete sonPoids;
42:     sonPoids = 0;
43:  }
44:
45:  int main()
46:  {
47:     Chat Frisky;
48:     cout << "Âge de Frisky : " << Frisky.GetAge() << endl;
49:     cout << "L'âge de Frisky est fixé à 6 ans...\n";
50:     Frisky.SetAge(6);
51:     cout << "Création de l'objet Boots à partir de Frisky\n";
52:     Chat Boots(Frisky);
53:     cout << "Âge de Frisky : " << Frisky.GetAge() << endl;
54:     cout << "Âge de Boots : " << Boots.GetAge() << endl;
55:     cout << "L'âge de Frisky est fixé à 7 ans...\n";
56:     Frisky.SetAge(7);
57:     cout << "Âge de Frisky : " << Frisky.GetAge() << endl;
58:     cout << "Âge de Boots : " << Boots.GetAge() << endl;
59:     return 0;
60:  }
```

Ce programme produit le résultat suivant :

```
Âge de Frisky : 5
L'âge de Frisky est fixé à 6 ans...
Création de l'objet Boots à partir de Frisky
Âge de Frisky : 6
Âge de Boots : 6
L'âge de Frisky est fixé à 7 ans...
Âge de Frisky : 7
Âge de Boots : 6
```

La classe `Chat` est déclarée de la ligne 6 à la ligne 19. Vous pouvez constater qu'un constructeur par défaut et un constructeur de copie sont déclarés respectivement à la ligne 9 et à la ligne 10. On reconnaît que le constructeur de la ligne 10 est un constructeur de copie, car il prend en paramètre une référence (une référence de constante dans ce cas) à un objet du même type.

Les deux variables membres déclarées aux lignes 17 et 18 sont chacune associées à un pointeur sur un entier. En règle générale, les variables membres sont rarement stockées comme des pointeurs, mais on l'a fait ici pour illustrer la gestion de variables membres sur le tas.

Le constructeur par défaut réserve de l'espace mémoire pour les deux variables entières sur le tas, puis leur affecte une valeur (lignes 21 à 27).

Le constructeur de copie commence à la ligne 29. Vous remarquerez que nous avons appelé son paramètre `rhs`, ce qui est assez courant en C++ (`rhs` est l'abréviation de *right-hand side*). Si vous examinez les lignes 33 et 34, vous remarquerez en effet que l'objet passé en paramètre se trouve à droite du signe égal.

Les lignes 31 à 32 allouent deux emplacements sur le tas, puis les deux lignes suivantes y affectent les valeurs des membres de l'objet `Chat` passé en paramètre.

Le paramètre `rhs` est un objet `Chat` passé au constructeur de copie en tant que référence constante. Étant un objet de la classe `Chat`, cette référence a les mêmes membres que n'importe quel autre objet `Chat`.

Tout objet `Chat` peut accéder à toutes les variables membres privées d'un autre objet `Chat` ; il est toutefois préférable d'utiliser les méthodes d'accès publiques dans la mesure du possible. La fonction membre `rhs.GetAge()` renvoie la valeur stockée dans l'emplacement mémoire pointé par la variable membre `sonAge` de `rhs`. Dans une véritable application, vous obtiendrez la valeur de `sonPoids` de la même manière, grâce une méthode d'accès. La ligne 34, toutefois, montre que les différents objets d'une même classe ont accès aux membres privés des autres. Une copie est alors réalisée directement depuis le membre `sonPoids` privé de l'objet `rhs`.

La Figure 10.3 montre ce qui s'est donc passé : les valeurs pointées par les variables membres de l'objet `Chat` passé en paramètre ont été copiées dans la mémoire allouée au nouvel objet `Chat`.

La ligne 47 crée l'objet `Frisky` de la classe `Chat`. Son âge apparaît à l'écran puis il reçoit la valeur 6 (ligne 50). À la ligne 52, le constructeur de copie duplique `Frisky` et baptise le nouvel objet `Boots`. Si `Frisky` avait été passé par valeur (et non par référence) à une fonction quelconque, le compilateur aurait appelé ce constructeur de copie de la même façon.

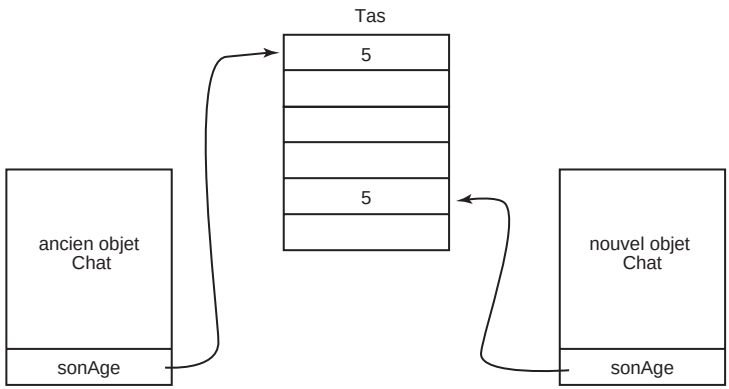


Figure 10.3
Illustration d'une copie en profondeur.

Les âges respectifs des deux objets `Chat` apparaissent alors à l'écran (lignes 53 et 54). `Boots` a 6 ans comme `Frisky`, puisqu'il a hérité des propriétés de ce dernier. En revanche, lorsque l'âge de `Frisky` est redéfini avec la valeur 7 (à la ligne 56), l'âge de `Boots` n'est pas modifié, ce qui prouve que les deux objets résident dans des zones distinctes en mémoire.

Lorsque les objets `Chat` deviennent hors de portée, ils sont effacés automatiquement par leurs destructeurs respectifs (lignes 37 à 43). Le destructeur d'un `Chat` utilise `delete` pour libérer les emplacements réservés sur le tas pour `sonAge` et `sonPoids` et affecte zéro à ces deux pointeurs pour plus de sécurité.

Surcharge des opérateurs

C++ dispose d'un certain nombre de types prédéfinis (`int`, `char`, `float`, etc.) et chacun d'eux possède un certain nombre d'opérateurs (`+`, `*`, etc.) permettant de réaliser des opérations mathématiques ou logiques. Vous pouvez également ajouter ces opérateurs à vos propres classes.

La surcharge d'opérateur est une technique que nous allons étudier dans le Listing 10.6. Nous allons créer une classe `Compteur` qui servira au comptage (quelle surprise !) dans les boucles et les autres situations où l'on a besoin d'incrémenter, décrémenter ou tester un nombre.

Listing 10.6 : Implémentation de la classe *Counter*

```
1:  // Listing 10.6 - La classe Compteur
2:
3:  #include <iostream>
4:  using namespace std;
```

```
 5:
 6:  class Compteur
 7:  {
 8:  public:
 9:     Compteur();
10:     ~Compteur(){}
11:     int GetVal()const { return saVal; }
12:     void SetVal(int x) {saVal = x; }
13:
14:  private:
15:     int saVal;
16:  };
17:
18:  Compteur::Compteur():
19:    saVal(0)
20:  {}
21:
22:  int main()
23:  {
24:     Compteur i;
25:     cout << "La valeur de i est " << i.GetVal() << endl;
26:     return 0;
27:  }
```

Ce programme produit le résultat suivant :

```
La valeur de i est 0
```

Telle qu'elle est écrite, cette classe est assez inutile. Déclarée de la ligne 6 à la ligne 16, elle ne contient qu'une variable membre de type `int`. Le constructeur par défaut, déclaré à la ligne 9, initialise l'unique variable membre `saVal` à zéro (ligne 18).

À la différence d'un objet standard de type `int`, l'objet `Compteur` ne peut pas être incrémenté, décrémenté, additionné, etc. En outre, l'affichage de sa valeur est bien plus compliqué que celle d'un entier !

Implémentation d'une fonction d'incrémentation

La surcharge d'opérateurs va permettre de restituer un certain nombre des fonctionnalités qui ont été supprimées de cette classe. Il y a deux moyens, par exemple, d'ajouter la possibilité d'incrémenter un objet `Compteur`. Le premier consiste à écrire une méthode d'incrémentation, comme dans le Listing 10.7.

Listing 10.7 : Ajout d'un opérateur d'incrémentation

```
 1:  // Listing 10.7 - La classe Compteur
 2:
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  class Compteur
 7:  {
 8:  public:
 9:     Compteur();
10:     ~Compteur(){}
11:     int GetVal() const { return saVal; }
12:     void SetVal(int x) {saVal = x; }
13:     void Increment() { ++saVal; }
14:
15:  private:
16:     int saVal;
17:  };
18:
19:  Compteur::Compteur():
20:    saVal(0)
21:  {}
22:
23:  int main()
24:  {
25:     Compteur i;
26:     cout << "La valeur de i est " << i.GetVal() << endl;
27:     i.Increment();
28:     cout << "La valeur de i est " << i.GetVal() << endl;
29:     return 0;
30:  }
```

Ce programme produit le résultat suivant :

```
La valeur de i est 0
La valeur de i est 1
```

La fonction `Increment` est définie à la ligne 13. Bien qu'elle fonctionne, elle est assez lourde à utiliser. Il serait ici plus judicieux d'ajouter un opérateur ++.

Surcharger l'opérateur préfixe

Les opérateurs préfixes peuvent être surchargés à l'aide d'une déclaration de la forme :

```
Type_de_valeur_renvoyee operatorop()
```

Ici, *op* correspond à l'opérateur qui doit être surchargé. L'opérateur ++ préfixe peut donc être surchargé en utilisant la syntaxe suivante :

```
void operator++()
```

Cette instruction indique que vous surchargez l'opérateur ++ et qu'il ne renverra pas de résultat – c'est la raison pour laquelle le type du résultat est `void`. Le Listing 10.8 modifie la classe précédente pour intégrer cette nouvelle fonctionnalité.

Listing 10.8 : Surcharge de l'opérateur ++

```
// Listing 10.8 - La classe Compteur
// opérateur préfixe d'incrémentation

#include <iostream>
using namespace std;

class Compteur
{
public:
   Compteur();
   ~Compteur(){}
   int GetVal()const { return saVal; }
   void SetVal(int x) {saVal = x; }
   void Increment() { ++saVal; }
   void operator++ () { ++saVal; }

private:
   int saVal;
};

Compteur::Compteur ():
  saVal(0)
{}

int main()
{
   Compteur i;
   cout << "La valeur de i est " << i.GetVal() << endl;
   i.Increment();
   cout << "La valeur de i est " << i.GetVal() << endl;
   ++i;
   cout << "La valeur de i est " << i.GetVal() << endl;
   return 0;
}
```

Ce programme produit le résultat suivant :

```
La valeur de i est 0
La valeur de i est 1
La valeur de i est 2
```

La ligne 15 surcharge `operator++`. Vous pouvez constater que cette surcharge incrémente simplement la valeur du membre privé `saVal`. Il est ensuite utilisé à la ligne 31 et vous remarquerez que cette syntaxe est bien plus proche de celle d'un type prédéfini comme `int`.

Vous pourriez également en profiter pour ajouter les fonctionnalités supplémentaires pour lesquelles on a été amené à créer une classe `Compteur`, comme la détection du dépassement de capacité. Cela dit, cette écriture de l'opérateur d'incrémentation recèle une erreur car, si vous l'utilisez à droite d'une expression d'affectation, vous obtiendrez une erreur. Voici un exemple :

```
Compteur a = ++i;
```

Cette instruction est supposée créer un nouvel objet `a` de la classe `Compteur`, puis lui affecter la valeur de `i` après son incrémentation. Le constructeur de copie par défaut va gérer cette affectation, mais votre opérateur d'incrémentation renverra `void`, pas un objet `Compteur`. Or, il est interdit d'affecter la valeur `void` à quoi que ce soit (on ne peut rien créer à partir de rien).

Renvoi de valeurs à partir des opérateurs surchargés

Votre objectif est de renvoyer un objet `Compteur` qui pourra être affecté à un autre objet de cette classe. Quel objet devez-vous récupérer après traitement ? Une solution consiste à créer un objet temporaire dans la fonction, puis à le renvoyer, comme dans le Listing 10.9.

Listing 10.9 : Renvoi d'un objet temporaire

```
 1:  // Listing 10.9 - operator++ renvoie un objet temporaire
 2:
 3:  #include <iostream>
 4:
 5:  using namespace std;
 6:
 7:  class Compteur
 8:  {
 9:  public:
10:     Compteur ();
11:     ~Compteur (){}
12:     int GetVal()const { return itsVal; }
```

```
13:     void SetVal(int x) {saVal = x; }
14:     void Increment() { ++saVal; }
15:     Compteur operator++();
16:
17:  private:
18:     int saVal;
19:
20:  };
21:
22:  Compteur::Compteur():
23:    saVal(0)
24:  {}
25:
26:  Compteur Compteur::operator++()
27:  {
28:     ++saVal;
29:     Compteur temp;
30:     temp.SetVal(saVal);
31:     return temp;
32:  }
33:
34:  int main()
35:  {
36:     Compteur i;
37:     cout << "La valeur de i est " << i.GetVal() << endl;
38:     i.Increment();
39:     cout << "La valeur de i est " << i.GetVal() << endl;
40:     ++i;
41:     cout << "La valeur de i est " << i.GetVal() << endl;
42:     Compteur a = ++i;
43:     cout << "Valeur de a : " << a.GetVal();
44:     cout << " et de i : " << i.GetVal() << endl;
45:     return 0;
46:  }
```

Ce programme produit le résultat suivant :

```
La valeur de i est 0
La valeur de i est 1
La valeur de i est 2
Valeur de a : 3 et de i : 3
```

Ici, la fonction `operator ++` déclarée à la ligne 15 et définie aux lignes 26 à 32 renvoie un objet `Compteur`. Pour cela, on crée la variable temporaire `temp` à la ligne 29 afin d'accueillir la valeur de l'objet à incrémenter. Après l'incrémentation, cette valeur temporaire est renvoyée. À la ligne 42, vous pouvez constater que la variable temporaire est immédiatement affectée à `a`.

Renvoi de variables temporaires anonymes

Il n'est pas toujours utile d'attribuer un nom à une variable temporaire comme celle de la ligne 29. Si `Compteur` avait un constructeur qui prenait une valeur en paramètre, vous auriez pu simplement renvoyer le résultat de ce constructeur comme valeur de retour de l'opérateur d'incrémentation. C'est ce que nous faisons dans le Listing 10.10.

Listing 10.10 : Renvoi d'un objet temporaire anonyme

```
 1:  // Listing 10.10 - operator++ renvoie un objet temporaire anonyme
 2:
 3:  #include <iostream>
 4:
 5:  using namespace std;
 6:
 7:  class Compteur
 8:  {
 9:  public:
10:     Compteur ();
11:     Compteur (int val);
12:     ~Compteur (){}
13:     int GetVal()const { return saVal; }
14:     void SetVal(int x) {saVal = x; }
15:     void Increment() { ++saVal; }
16:     Compteur operator++ ();
17:
18:  private:
19:     int saVal;
20:  };
21:
22:  Compteur::Compteur ():
23:    saVal(0)
24:  {}
25:
26:  Compteur::Compteur (int val):
27:    saVal(val)
28:  {}
29:
30:  Compteur Compteur::operator++()
31:  {
32:     ++saVal;
33:     return Compteur(saVal);
34:  }
35:
36:  int main()
37:  {
38:     Compteur i;
```

```
39:     cout << "La valeur de i est " << i.GetVal() << endl;
40:     i.Increment();
41:     cout << "La valeur de i est " << i.GetVal() << endl;
42:     ++i;
43:     cout << "La valeur de i est " << i.GetVal() << endl;
44:     Compteur a = ++i;
45:     cout << "Valeur de a : " << a.GetVal();
46:     cout << " et de i : " << i.GetVal() << endl;
47:     return 0;
48: }
```

Ce programme produit le résultat suivant :

```
La valeur de i est 0
La valeur de i est 1
La valeur de i est 2
Valeurs de a : 3 et de i : 3
```

Le nouveau constructeur de la ligne 11 attend un paramètre de type `int`. Il est implémenté aux lignes 26 à 28 et initialise la variable `saVal` avec la valeur qu'on lui a transmise.

Vous pouvez constater que l'implémentation d'`operator++` est simplifiée. En effet, `saVal` est incrémentée à la ligne 32 puis la ligne suivante crée un objet `Compteur` temporaire à partir de la nouvelle valeur de `saVal`. Cet objet est renvoyé comme résultat de `operator++`.

Cette procédure est plus élégante, mais elle soulève la question suivante : "ne pourrait-on pas se passer des objets temporaires ?". N'oublions pas que chaque objet temporaire doit être créé puis supprimé et que ces deux opérations peuvent être coûteuses. En outre, l'objet `i` existe déjà et a déjà la bonne valeur : pourquoi ne pas le renvoyer ? Ce problème peut être résolu grâce au pointeur `this`.

Utilisation du pointeur *this*

Le pointeur `this` est automatiquement passé à toutes les fonctions membres, y compris les opérateurs surchargés comme `operator++()`. Dans les listings que nous avons montrés, `this` pointe vers `i`. Une fois déréférencé, il renvoie l'objet `i` qui a déjà la bonne valeur dans sa variable membre `itsVal`. Le Listing 10.11 illustre le renvoi du pointeur `this` déréférencé afin d'économiser la création d'un objet temporaire superflu.

Listing 10.11 : Renvoi du pointeur *this*

```
1:  // Listing 10.11 - Renvoi du pointeur this déréférencé
2:
3:  #include <iostream>
```

```
 4:
 5:  using namespace std;
 6:
 7:  class Compteur
 8:  {
 9:  public:
10:     Compteur();
11:     ~Compteur(){}
12:     int GetVal()const { return saVal; }
13:     void SetVal(int x) {saVal = x; }
14:     void Increment() { ++saVal; }
15:     const Compteur& operator++();
16:
17:  private:
18:     int saVal;
19:  };
20:
21:  Compteur::Compteur():
22:    saVal(0)
23:  {};
24:
25:  const Compteur& Compteur::operator++()
26:  {
27:     ++saVal;
28:     return *this;
29:  }
30:
31:  int main()
32:  {
33:     Compteuri;
34:     cout << "La valeur de i est " << i.GetVal() << endl;
35:     i.Increment();
36:     cout << "La valeur de i est " << i.GetVal() << endl;
37:     ++i;
38:     cout << "La valeur de i est " << i.GetVal() << endl;
39:     Compteur a = ++i;
40:     cout << "Valeur de a : " << a.GetVal();
41:     cout << " et de i : " << i.GetVal() << endl;
42:     return 0;
43:  }
```

Ce programme produit le résultat suivant :

```
La valeur de i est 0
La valeur de i est 1
La valeur de i est 2
Valeur de a : 3 et de i : 3
```

Dans ce programme, `operator++` déréférence le pointeur `this` afin de renvoyer l'objet courant (lignes 25 à 29). Le résultat de cette opération est donc un objet `Compteur` qui peut être affecté à la variable `a`. Comme nous l'avons déjà évoqué, si l'objet `Compteur` allouait de la mémoire, il faudrait surcharger le constructeur de copie par défaut mais, ici, il suffit amplement.

La valeur renvoyée est une référence à un objet `Compteur`, ce qui vous épargne d'avoir à créer un objet temporaire supplémentaire. Cette référence est constante car la valeur ne doit pas pouvoir être modifiée par la fonction qui utilisera le `Compteur` renvoyé.

L'objet `Compteur` renvoyé doit être constant. Dans le cas contraire, il serait possible d'en changer la valeur. S'il n'était pas constant, vous pourriez par exemple écrire la ligne 39 de la façon suivante :

```
39:     Compteur a = ++++i;
```

On s'attend alors à ce que l'opérateur d'incrémentation (++) soit appelé sur le résultat de l'appel de `++i`. Cela reviendrait en fait à appeler deux fois l'opérateur d'incrémentation sur l'objet `i`, ce qu'il est généralement conseillé d'éviter.

Faites un essai : remplacez la valeur renvoyée pour qu'elle ne soit plus constante à la fois dans la déclaration et l'implémentation (lignes 15 et 25), puis remplacez la ligne 39 par (`++++i`). Placez un point d'arrêt dans votre débogueur à la ligne 39 et effectuez un pas à pas détaillé (*step into*). Vous constaterez que l'opérateur d'incrémentation est appelé deux fois. Il est appliqué à la valeur renvoyée (devenue non constante).

C'est pour éviter ce type de comportement que vous devez déclarer la valeur renvoyée comme constante. Si vous remettez les lignes 15 et 25 en l'état (`const`), sans modifier la ligne 39 (`++++i`) le compilateur indiquera que vous ne pouvez pas appeler l'opérateur d'incrémentation sur un objet constant.

Surcharger l'opérateur suffixe

S'il est possible de surcharger un opérateur préfixe (voir plus haut), pourquoi ne pourrions-nous pas également surcharger l'opérateur suffixe d'incrémentation ? Comment le compilateur va-t-il distinguer le préfixe du suffixe ? Par convention, il suffit d'indiquer un paramètre entier dans la déclaration de l'opérateur. La valeur qu'il contient ne sera pas prise en compte, mais il servira à signaler qu'il s'agit d'un opérateur suffixe.

Différence entre opérateurs préfixe et suffixe

Les deux opérateurs préfixe et suffixe ont été étudiés au Chapitre 4 (voir Listing 4.3).

Vous avez vu que préfixe signifie "on incrémente puis on utilise", alors que suffixe signifie "on utilise puis on incrémente".

L'opérateur préfixe peut donc se contenter d'incrémenter la valeur puis de renvoyer l'objet modifié alors que l'opérateur suffixe doit renvoyer la valeur telle qu'elle était avant l'incrémentation. Pour ce faire, il faut donc créer un objet temporaire qui contiendra la valeur initiale, incrémenter ensuite la valeur de l'objet original puis renvoyer l'objet temporaire.

Examinez la ligne suivante :

```
a = x++;
```

Si la x valait 5, a aura la valeur 5 et x la valeur 6 après l'exécution de cette ligne. La valeur stockée dans x a donc bien été renvoyée et attribuée à a avant d'être incrémentée. Si x est un objet, son opérateur d'incrémentation suffixe doit enregistrer la valeur initiale (5) dans un objet temporaire, affecter à x la valeur 6, puis renvoyer l'objet temporaire pour attribuer sa valeur d'origine à a.

Comme on renvoie l'objet temporaire, celui-ci doit être renvoyé par valeur et non par référence puisque sa portée se limite à la fonction.

Les opérateurs préfixe et suffixe sont implémentés dans le Listing 10.12.

Listing 10.12 : Opérateurs préfixe et suffixe

```
 1:  // Listing 10.12 - Opérateurs préfixe et suffixe
 2:
 3:  #include <iostream>
 4:
 5:  using namespace std;
 6:
 7:  class Compteur
 8:  {
 9:  public:
10:     Compteur();
11:     ~Compteur(){}
12:     int GetVal()const { return saVal; }
13:     void SetVal(int x) {saVal = x; }
14:     const Compteur& operator++ ();      // préfixe
15:     const Compteur operator++ (int);    // suffixe
16:
17:  private:
18:     int saVal;
19:  };
20:
21:  Compteur::Compteur():
22:    itsVal(0)
23:  {}
24:
25:  const Compteur& Compteur::operator++()
```

```
26:  {
27:     ++saVal;
28:     return *this;
29:  }
30:
31:  const Compteur Compteur::operator++(int flag)
32:  {
33:     Compteur temp(*this);
34:     ++saVal;
35:     return temp;
36:  }
37:
38:  int main()
39:  {
40:     Compteur i;
41:     cout << "La valeur de i est " << i.GetVal() << endl;
42:     i++;
43:     cout << "La valeur de i est " << i.GetVal() << endl;
44:     ++i;
45:     cout << "La valeur de i est " << i.GetVal() << endl;
46:     Compteur a = ++i;
47:     cout << "Valeur de a : " << a.GetVal();
48:     cout << " et de i : " << i.GetVal() << endl;
49:     a = i++;
50:     cout << "Valeur de a : " << a.GetVal();
51:     cout << " et de i : " << i.GetVal() << endl;
52:     return 0;
53:  }
```

Ce programme produit le résultat suivant :

```
La valeur de i est 0
La valeur de i est 1
La valeur de i est 2
Valeur de a : 3 et de i : 3
Valeur de a : 3 et de i : 4
```

Déclaré à la ligne 15, l'opérateur suffixe est implémenté de la ligne 31 à la ligne 36. L'opérateur préfixe est déclaré à la ligne 14.

Le paramètre passé à l'opérateur suffixe à la ligne 32 (`flag`) sert simplement à indiquer qu'il s'agit de l'opérateur suffixe, mais il n'est jamais utilisé.

Surcharge des opérateurs mathématiques binaires

L'opérateur d'incrémentation est un opérateur unaire, car il n'agit que sur un objet. La plupart des opérateurs mathématiques sont binaires ; ils portent sur deux objets (un de

la classe courante et l'autre de n'importe quelle classe). La surcharge d'opérateurs comme les opérateurs d'addition (+), de soustraction (–), de multiplication (*), de division (/) et de modulo (%) sera donc évidemment différente de la surcharge des opérateurs de préfixe et de suffixe. Voyons, par exemple, comment surcharger l'opérateur + pour les objets `Compteur`.

L'objectif est de pouvoir déclarer deux variables `Compteur`, puis de les additionner, comme ici :

```
Compteur VarUne, VarDeux, VarTrois;
VarTrois = VarUne + VarDeux;
```

Là encore, on peut commencer par écrire une méthode `Add()` qui prendra un `Compteur` en paramètre, ajoutera les valeurs et renverra un `Compteur` correspondant au résultat de l'addition. C'est cette approche que nous utilisons dans le Listing 10.13.

Listing 10.13 : La méthode *Add()*

```
 1: // Listing 10.13 - Méthode Add
 2:
 3: #include <iostream>
 4:
 5: using namespace std;
 6:
 7: class Compteur
 8: {
 9: public:
10:    Compteur();
11:    Compteur(int ValInit);
12:    ~Compteur(){}
13:    int GetVal()const { return saVal; }
14:    void SetVal(int x) {saVal = x; }
15:    Compteur Add(const Compteur&);
16:
17: private:
18:    int saVal;
19: };
20:
21: Compteur::Compteur(int ValInit):
22:   saVal(ValInit)
23: {}
24:
25: Compteur::Compteur():
26:   saVal(0)
27: {}
28:
```

```
29:  Compteur::Add(const Compteur & rhs)
30:  {
31:     return Compteur(saVal+ rhs.GetVal());
32:  }
33:
34:  int main()
35:  {
36:     Compteur varUne(2), varDeux(4), varTrois;
37:     varTrois = varUne.Add(varDeux);
38:     cout << "varUne : " << varUne.GetVal()<< endl;
39:     cout << "varDeux : " << varDeux.GetVal() << endl;
40:     cout << "varTrois : " << varTrois.GetVal() << endl;
41:
42:   return 0;
43:  }
```

Ce programme produit le résultat suivant :

```
varUne : 2
varDeux : 4
varTrois : 6
```

Déclarée à la ligne 15, la fonction `Add()` prend en paramètre une référence constante au `Compteur` correspondant au nombre à additionner à l'objet en cours. Elle renvoie un objet `Compteur` qui sera affecté à la partie gauche de l'affectation réalisée à la ligne 37. `varUne` et `varDeux` sont respectivement l'objet et le paramètre passés à la fonction `Add()`, le résultat de l'opération étant affecté à `varTrois`

Pour créer `varTrois` sans devoir l'initialiser, vous avez besoin d'un constructeur par défaut. Celui-ci initialise `saVal` à zéro (lignes 25 et 27). En revanche, `varOne` et `varTwo` doivent être initialisées avec une valeur non nulle, ce qui oblige à créer un autre constructeur (lignes 21 à 23). Une autre solution au problème serait de passer une valeur de zéro par défaut au constructeur déclaré à la ligne 11.

Les lignes 29 à 32 réalisent l'addition des deux objets. La méthode fonctionne, mais son utilisation n'est pas très naturelle.

Surcharge de l'opérateur d'addition (opérateur *+*)

La surcharge de l'opérateur + permet d'utiliser la classe `Compteur` de façon plus naturelle. On rappelle que, pour surcharger un opérateur, il faut utiliser la syntaxe suivante :

```
TypeRetour operator op()
```

Le Listing 10.4 surcharge l'opérateur d'addition en utilisant cette fonctionnalité.

Listing 10.14 : Surcharge de l'opérateur +

```
// Listing 10.14 - Surcharge de l'opérateur +

#include <iostream>

using namespace std;

class Compteur
{
public:
   Compteur();
   Compteur(int valInit);
   ~Compteur (){}
   int GetVal()const {return saVal; }
   void SetVal(int x) {saVal = x; }
   Compteur operator+ (const Compteur&);
private:
   int saVal;
};

Compteur::Compteur(int valInit):
  saVal(valInit)
{}

Compteur::Compteur ():
  saVal(0)
{}

Compteur Compteur::operator+(const Compteur & rhs)
{
   return Compteur(saVal + rhs.GetVal());
}

int main()
{
   Compteur varUne(2), varDeux(4), varTrois;
   varTrois = varUne + varDeux;
   cout << "varUne : " << varUne.GetVal()<< endl;
   cout << "varDeux : " << varDeux.GetVal() << endl;
   cout << "varTrois : " << varTrois.GetVal() << endl;

   return 0;
}
```

Ce programme produit le résultat suivant :

```
varUne : 2
varDeux : 4
varTrois : 6
```

`operator+` est déclaré à la ligne 15 et défini de la ligne 28 à la ligne 31.

On remarque que cela ressemble beaucoup à la déclaration et la définition de la fonction `Add()` du programme précédent. En revanche, leur syntaxe d'utilisation est radicalement différente. Il est bien plus naturel d'écrire :

```
varTrois = varUne + varDeux;
```

que :

```
varTrois = varUne.Add(varDeux);
```

Les deux expressions donnent le même résultat, mais la première est plus facile à comprendre et à utiliser.

Ce nouvel opérateur est utilisé à la ligne 36 :

```
36:    varTrois = varUne + varDeux;
```

Ce qui est traduit par le compilateur en :

```
varTrois = varUne.operator+(varDeux);
```

(vous auriez d'ailleurs pu l'écrire également sous cette forme).

La méthode `operator+()` est appelée sur l'opérande gauche et reçoit l'opérande droit en paramètre.

Pour en savoir plus sur la surcharge d'opérateurs

Les opérateurs surchargés peuvent être des méthodes membres, comme on vient de le montrer, ou des fonctions standard. Cette deuxième possibilité sera décrite au Chapitre 15 lorsque nous aborderons les fonctions amies.

Les seuls opérateurs qui doivent être membres d'une classe sont l'opérateur d'affectation (`=`), l'opérateur d'indexation (`[]`), l'opérateur d'appel de fonction (`()`) et l'opérateur d'indirection (`->`).

L'opérateur `[]` sera présenté au Chapitre 13, alors que la surcharge de l'opérateur `->` sera traitée au Chapitre 15 lorsque nous présenterons les pointeurs intelligents.

Restrictions sur la surcharge des opérateurs

Les opérateurs des types prédéfinis (comme `int`) ne peuvent pas être surchargés. Par ailleurs, leur priorité ne peut pas être modifiée, pas plus que leur arité (unaire ou binaire). Vous ne pouvez pas non plus créer de nouveaux opérateurs : il est par exemple impossible de créer un opérateur `**` pour implémenter l'opération "puissance de".



L'arité d'un opérateur indique le nombre de termes utilisés avec celui-ci. Certains opérateurs C++ sont unaires et agissent sur un seul terme (`maValeur++`). D'autres sont binaires et portent sur deux termes (`a+b`). Il n'existe qu'un seul opérateur ternaire (trois termes), l'opérateur `?` et c'est la raison pour laquelle on le nomme souvent tout simplement l'opérateur ternaire (`a > b ? x : y`).

Savoir utiliser la surcharge

La surcharge d'opérateur est l'une des fonctionnalités dont abusent le plus les programmeurs C++ débutants. Il est en effet tentant de définir de nouvelles fonctionnalités pour certains des opérateurs les plus obscurs, mais cela a pour effet irrémédiable de rendre le code confus et difficile à relire.

Il est peut être amusant de transformer les opérateurs, en décidant par exemple que l'opérateur `+` et l'opérateur `*` permettront d'effectuer, respectivement, des divisions et des additions, mais aucun développeur sérieux ne s'y risquerait. Le plus grand danger vient d'une utilisation bien intentionnée mais originale des opérateurs – utiliser `+` pour effectuer une concaténation de chaîne ou `/` pour découper une chaîne, par exemple. Il y a de bonnes raisons de penser à ces utilisations, mais il y en a encore de meilleures pour s'en méfier. N'oubliez pas que le but de la surcharge des opérateurs est d'améliorer la lisibilité du code et de faciliter l'utilisation des objets.

Faire	*Ne pas faire*
● Surcharger les opérateurs pour simplifier la mise en œuvre et la maintenance d'un programme. ● Renvoyer un objet de la classe à partir des opérateurs surchargés.	● Créer des opérateurs arithmétiques "non intuitifs". ● Confondre les opérateurs de préfixe et de suffixe, en particulier lors d'une surcharge.

L'opérateur d'affectation

La quatrième et dernière méthode fournie par le compilateur, si vous ne la définissez pas vous-même, est l'opérateur d'affectation `operator=()`. Cet opérateur est appelé à chaque fois que vous affectez une valeur à un objet. Voici un exemple :

```
Chat chatUn(5, 7);
Chat chatDeux(3, 4);
// ... autres instructions
chatDeux = chatUn;
```

L'objet `chatUn` est créé et initialisé avec `sonAge` égal à 5 et `sonPoids` égal à 7. Un autre objet, `chatDeux`, est créé et initialisé avec les valeurs 3 et 4.

Après un certain nombre d'opérations, les valeurs de `chatUn` sont affectées à `chatDeux`. Deux problèmes se posent ici : que se passe-t-il si `sonAge` est un pointeur et que deviennent les valeurs initiales de `chatDeux` ?

La gestion des variables membres qui stockent leurs valeurs sur le tas a été évoquée plus haut, lors de la présentation du constructeur de copie. Les mêmes problèmes risquent de survenir à l'issue de cette opération, comme on l'a vu avec les Figures 10.1 et 10.2.

En C++, il faut faire la différence entre une copie de surface et une copie en profondeur. La première copie simplement les membres et les deux objets pointent finalement sur le même emplacement du tas, alors que la seconde alloue la mémoire nécessaire pour y dupliquer les contenus des objets, comme on l'a vu avec la Figure 10.3.

Cependant, un autre problème se pose avec l'opérateur d'affectation. L'objet `chatDeux` existe déjà et sa mémoire est déjà allouée lorsque l'affectation est réalisée. Pour éviter toute fuite mémoire, il faut donc supprimer l'emplacement qu'il occupait, mais que se passe-t-il alors si l'on affecte `chatDeux` à lui-même, comme ici ?

```
chatDeux = chatDeux;
```

Cette opération est le plus souvent le résultat d'une erreur. Elle peut également être accidentelle et découler de référencements et de déréférencements successifs de pointeurs cachant cette auto-affectation.

Si vous ne traitez pas correctement ce problème, `chatDeux` libérera l'espace qu'il occupe en mémoire. Il se posera alors un gros problème lorsqu'il sera prêt à recevoir les valeurs de la partie droite de l'affectation puisqu'elles auront disparu !

Pour éviter cela, l'opérateur d'affectation doit tester si la partie droite de l'opérateur d'affectation est l'objet lui-même. Pour cela, il examine la valeur du pointeur `this`, comme le montre le Listing 10.15. Ceci évite également le problème que nous venons d'évoquer.

Listing 10.15 : Opérateur d'affectation

```
 1: // Listing 10.15 - L'opérateur d'affectation
 2:
 3: #include <iostream>
 4:
 5: using namespace std;
 6:
 7: class Chat
 8: {
 9: public:
10:    Chat();              // constructeur par défaut
11:    // constructeur de copie et destructeur absents !
12:    int GetAge() const { return *sonAge; }
13:    int GetPoids() const { return *sonPoids; }
14:    void SetAge(int age) { *sonAge = age; }
15:    Chat & operator=(const Chat &);
16:
17: private:
18:    int *sonAge;
19:    int *sonPoids;
20: };
21:
22: Chat::Chat()
23: {
24:    sonAge = new int;
25:    sonPoids = new int;
26:    *sonAge = 5;
27:    *sonPoids = 9;
28: }
29:
30:
31: Chat & Chat::operator=(const Chat & rhs)
32: {
33:    if (this == &rhs)
34:      return *this;
35:    *sonAge = rhs.GetAge();
36:    *sonPoids = rhs.GetPoids();
37:    return *this;
38: }
39:
40:
41: int main()
42: {
43:    Chat Frisky;
44:    cout << "Âge de Frisky : " << Frisky.GetAge() << endl;
45:    cout << "L'âge de Frisky est fixé à 6...\n";
```

```
46:     Frisky.SetAge(6);
47:     Chat Minou;
48:     cout << "Âge de Minou : " << Minou.GetAge() << endl;
49:     cout << "Copie de Frisky dans Minou...\n";
50:     Minou = Frisky;
51:     cout << "Âge de Minou : " << Minou.GetAge() << endl;
52:     return 0;
53: }
```

Ce programme produit le résultat suivant :

```
Âge de Frisky : 5
L'âge de Frisky est fixé a 6...
Âge de Minou : 5
Copie de Frisky dans Minou...
Âge de Minou : 6
```

Ce programme reprend à nouveau la classe `Chat`, mais ne définit ni le constructeur de copie ni le destructeur afin gagner de la place. L'opérateur d'affectation surchargé est déclaré à la ligne 15 et défini de la ligne 31 à la ligne 38.

La ligne 33 compare les deux parties de l'affectation. Pour cela, on teste si l'adresse de l'objet `Chat` de droite (`rhs`) est égale à l'adresse stockée dans le pointeur `this`. Si elles sont identiques, il n'y a rien à faire, car l'objet de gauche est identique à l'objet de droite. Du coup, la ligne 34 renvoie l'objet courant.

Si l'objet de droite est différent, les membres sont copiés dans l'objet aux lignes 35 et 36 avant qu'il soit renvoyé.

L'utilisation de l'opérateur d'affectation est illustrée à la ligne 50 du programme principal, lorsqu'un objet `Chat` appelé `Frisky` est affecté à l'objet `Chat Minou`. Le reste devrait vous être familier.

Ce listing suppose que deux objets pointant vers la même adresse sont identiques. Bien entendu, il est également possible de surcharger l'opérateur d'égalité (==) afin de définir ce que vous entendez par "égalité" de deux objets de cette classe.

Conversion de type (transtypage)

Maintenant que vous savez affecter un objet à un autre objet du même type, considérons une autre situation. Que se passera-t-il si vous tentez d'affecter une variable d'un type prédéfini (`int` ou `unsigned short`, par exemple) à un objet d'une classe définie par l'utilisateur ? Prenons l'exemple de la classe `Compteur` créée précédemment : que

se passera-t-il si vous lui affectez un entier ? Le Listing 10.16 tente d'effectuer cette opération.



Ce programme ne se compilera pas !

Listing 10.16 : Tentative d'affectation d'un *int* à un *Compteur*

```
// Listing 10.16 - Ce code provoquera une erreur de compilation !

#include <iostream>

using namespace std;

class Compteur
{
public:
   Compteur();
   ~Compteur(){}
   int GetVal()const {return saVal; }
   void SetVal(int x) {saVal = x; }
private:
   int saVal;
};

Compteur::Compteur ():
  saVal(0)
{}

int main()
{
   int unInt = 5;
   Compteur unCpt = unInt;
   cout << "unCpt : " << unCpt.GetVal() << endl;
   return 0;
}
```

La compilation de ce programme produira le message suivant :

```
Compiler error! Unable to convert int to Compteur
```

En d'autres termes, le compilateur est incapable de convertir un type prédéfini en un objet d'une classe définie par l'utilisateur.

Déclarée de la ligne 7 à la ligne 16, la classe `Compteur` n'a qu'un constructeur par défaut. Elle ne déclare aucune méthode permettant de transformer un type prédéfini en `Compteur`. La ligne 24 de la fonction `main()` déclare un entier qui est ensuite affecté à un objet `Compteur`, mais cette ligne produit une erreur de compilation. En effet, le compilateur est incapable de savoir comment affecter un `int` à la variable membre `saVal`, sauf si vous lui expliquez.

Le Listing 10.17 corrige le problème en ajoutant un opérateur de conversion, c'est-à-dire un constructeur prenant un `int` en paramètre et renvoyant un objet `Compteur`.

Listing 10.17 : Conversion d'un objet de type *int* en objet de la classe *Compteur*

```
 1:  // Listing 10.17 - Constructeur/opérateur de conversion
 2:
 3:  #include <iostream>
 4:
 5:  using namespace std;
 6:
 7:  class Compteur
 8:  {
 9:  public:
10:     Compteur();
11:     Compteur(int val);
12:     ~Compteur(){}
13:     int GetVal()const { return saVal; }
14:     void SetVal(int x) {saVal = x; }
15:  private:
16:     int saVal;
17:  };
18:
19:  Compteur::Compteur():
20:    saVal(0)
21:  {}
22:
23:  Compteur:: Compteur(int val):
24:    saVal(val)
25:  {}
26:
27:  int main()
28:  {
29:     int unInt = 5;
30:     Compteur unCpt = unInt;
31:     cout << "unCpt : " << unCpt.GetVal() << endl;
32:     return 0;
33:  }
```

Ce programme produit le résultat suivant :

```
unCpt : 5
```

Deux modifications majeures ont été apportées à ce programme. Le constructeur est surchargé pour pouvoir prendre en paramètre un objet de type `int` (ligne 11). Il est ensuite implémenté de la ligne 23 à la ligne 25. Ce constructeur permet de créer un objet `Compteur` à partir d'un `int`.

Le compilateur peut donc désormais appeler le constructeur en lui passant un `int` en paramètre :

Étape 1 : Création d'un compteur appelé `unCpt`.

C'est comme écrire `int x = 5;` pour créer une variable entière `x` et l'initialiser à la valeur `5`. Ici, nous créons un objet `Compteur` appelé `unCpt` et nous l'initialisons à l'aide de la variable entière `unInt`.

Étape 2 : Affectation de la valeur de `unInt` à `unCpt`

Cependant, `unInt` est un entier, pas un compteur ! Nous devons donc d'abord le convertir en `Compteur`. Le compilateur tentera automatiquement plusieurs conversions pour vous, mais vous devez lui indiquer comment faire en créant un constructeur. Ici, il faut donc un constructeur prenant un entier pour seul paramètre :

```
class Compteur
{
  Compteur(int x);
  // ..
};
```

Ce constructeur fabrique des objets `Compteur` à partir d'entiers en créant un compteur temporaire anonyme. Pour les besoins de notre explication, nous supposerons que l'objet `Compteur` temporaire créé à partir de l'entier court s'appelle `tempInt`.

Étape 3 : Affectation de `tempInt` à `unCpt`, ce qui est équivalent à :

```
unCpt = tempInt;
```

Ici, `tempInt` (le compteur temporaire créé par le constructeur) remplace ce qui se trouvait à droite de l'affectation. Le compilateur utilise maintenant le `Compteur` temporaire pour initialiser `unCpt`.

Pour mieux comprendre ce mécanisme, il faut savoir que *toutes* les surcharges d'opérateurs fonctionnent de la même façon : vous déclarez un opérateur surchargé à l'aide du

mot clé `operator`. Avec des opérateurs binaires (comme = ou +) la variable placée à droite devient le paramètre de ces surcharges. C'est ce que fait le constructeur. Par conséquent :

```
a = b;
```

devient :

```
a.operator=(b);
```

Pourtant, que se passerait-il si vous tentiez de remplacer l'affectation par les instructions suivantes ?

```
1:  Compteur unCpt(5);
2:  int unInt = unCpt;
3:  cout << "unInt : " << unInt  << endl;
```

Le compilateur produirait une erreur car, bien qu'il sache maintenant créer un `Compteur` à partir d'un `int`, il ne sait pas réaliser l'opération inverse.

Opérateurs de conversion

Pour gérer la conversion des objets de votre classe vers un autre type, C++ propose un certain nombre d'opérateurs de conversion que vous pouvez ajouter à vos classes afin qu'elles sachent se convertir implicitement vers des types prédéfinis. Le Listing 10.18 donne un exemple d'un tel opérateur. Vous pourrez remarquer que les opérateurs de conversion ne précisent pas le type de leur résultat, bien qu'ils renvoient une valeur convertie (en fait, le type du résultat est le nom même de la méthode de conversion).

Listing 10.18 : Conversion d'un objet *Compteur* en *unsigned short()*

```
 1:  // Listing 10.18 - Opérateurs de conversion
 2:  #include <iostream>
 3:
 4:  class Compteur
 5:  {
 6:  public:
 7:     Compteur();
 8:     Compteur(int val);
 9:     ~Compteur(){}
10:     int GetVal()const { return saVal; }
11:     void SetVal(int x) {saVal = x; }
12:     operator unsigned int();
13:  private:
14:     int saVal;
15:  };
16:
```

```
17:  Compteur::Compteur():
18:    saVal(0)
19:  {}
20:
21:  Compteur::Compteur(int val):
22:    saVal(val)
23:  {}
24:
25:  Compteur::operator unsigned int()
26:  {
27:     return ( int(saVal) );
28:  }
29:
30:  int main()
31:  {
32:     Compteur cpt(5);
33:     int unInt = cpt;
34:     std::cout << "unInt : " << unInt << std::endl;
35:     return 0;
36:  }
```

Ce programme produit le résultat suivant :

```
unInt : 5
```

La ligne 12 déclare l'opérateur de conversion. Cette déclaration commence par le mot-clé `operator` et ne n'indique pas le type du résultat. La fonction de conversion est implémentée de la ligne 25 à la ligne 28. La ligne 27 renvoie la valeur de `saVal` convertie en `int`.

Le compilateur étant en mesure de convertir un `int` en `Compteur` et *vice-versa*, les variables de ces deux types peuvent être affectées librement l'une à l'autre. Vous pouvez évidemment utiliser la même approche pour gérer également d'autres types.

Questions-réponses

Q Pourquoi utiliser des valeurs par défaut lorsqu'on peut surcharger une fonction ?

R Il est souvent plus aisé de gérer une seule fonction comprenant des valeurs par défaut que d'étudier le déroulement de deux fonctions. En outre, si vous modifiez une fonction surchargée, vous devez également mettre à jour toutes les autres fonctions surchargées correspondantes afin d'éviter toute erreur d'exécution ou de compilation.

Q Au vu de toutes ces contraintes, doit-on préférer les valeurs par défaut ?

R Non. Les fonctions surchargées offrent des fonctionnalités qui ne sont pas disponibles avec les valeurs par défaut. Par exemple, il est possible de modifier non seulement le

nombre de paramètres dans une liste, mais également leur type, ou encore de fournir une implémentation différente pour diverses combinaisons de types de paramètres.

Q Lors de la création d'un constructeur, comment les opérations doivent-elles se répartir entre la phase d'initialisation et la phase d'exécution ?

R En règle générale, il faut utiliser le plus possible la phase d'initialisation – c'est-à-dire initialiser toutes les variables membres à cet endroit. Certaines opérations, comme les calculs (y compris ceux qui servent à l'initialisation) et les affichages doivent avoir lieu dans le corps du constructeur.

Q Une fonction surchargée peut-elle inclure un paramètre par défaut ?

R Oui. Il est possible d'associer une liste de valeurs par défaut à une fonction surchargée. La syntaxe employée et les règles à respecter sont identiques à celles utilisées pour les variables par défaut de n'importe quelle fonction.

Q Pourquoi certaines fonctions membres sont-elles définies dans les déclarations de classe alors que d'autres ne le sont pas ?

R Les fonctions dont l'implémentation est définie lors de la déclaration d'une classe sont traduites par des fonctions en ligne. Généralement, cela ne concerne que les fonctions dont le corps est très simple. Vous pouvez également demander à ce qu'une méthode soit en ligne à l'aide du mot-clé `inline`, même si elle est définie en dehors de la déclaration de la classe.

Testez vos connaissances

1. En quoi les fonctions surchargées doivent-elles être différentes ?
2. Quelle est la différence entre une déclaration et une définition ?
3. Quand un constructeur de copie est-il appelé ?
4. Quand le destructeur est-il appelé ?
5. En quoi le constructeur de copie diffère-t-il de l'opérateur d'affectation (=) ?
6. Qu'est-ce que le pointeur `this` ?
7. Quelle est la différence entre la surcharge d'un opérateur préfixe d'incrémentation et la surcharge d'un opérateur suffixe ?
8. Est-il possible de surcharger l'opérateur + avec des entiers courts ?
9. Peut-on surcharger l'opérateur ++ pour qu'il décrémente un objet d'une classe ?
10. Quelle type de résultat doit figurer dans la déclaration des opérateurs de conversion ?

Exercices

1. Écrivez la déclaration d'une classe `Cercle` comprenant une variable membre `SonRayon`. Associez un constructeur par défaut, un destructeur et plusieurs méthodes d'accès à cette variable.
2. Dans la classe créée lors de l'exercice précédent, implémentez le constructeur par défaut, qui affecte la valeur 5 à la variable `sonRayon`. Faites-le dans la phase d'initialisation du constructeur et non dans le corps.
3. Ajoutez ensuite un deuxième constructeur prenant un paramètre permettant d'affecter une valeur à la variable `sonRayon`.
4. Dans la classe `Cercle`, créez un opérateur préfixe et un opérateur suffixe permettant d'incrémenter la variable `sonRayon`.
5. Modifiez la classe `Cercle` pour qu'elle stocke `sonRayon` sur le tas. Pour cela, vous devrez modifier les méthodes existantes.
6. Ajoutez un constructeur de copie à la classe `Cercle`.
7. Ajoutez un opérateur d'affectation à la classe `Cercle`.
8. Créez deux objets `Cercle`. Utilisez le constructeur par défaut pour le premier et affectez la valeur 9 au second. Appelez l'opérateur d'incrémentation pour chacun d'eux, puis affichez leurs valeurs. Enfin, après avoir affecté la valeur du second au premier, affichez leurs valeurs.
9. CHERCHEZ L'ERREUR : l'implémentation de l'opérateur d'affectation est erronée.

```
CARRE CARRE ::operator=(const CARRE & rhs)
{
      sonCote = new int;
      *sonCote = rhs.GetCote();
      return *this;
}
```

10. CHERCHEZ L'ERREUR : l'implémentation de l'opérateur d'addition est erronée.

```
EntierCourt  EntierCourt::operator+(const EntierCourt& rhs)
{
   saVal += rhs.GetVal();
   return *this;
}
```



11

Analyse et conception orientée objet

Au sommaire de ce chapitre

- Comment utiliser l'analyse orientée objet pour comprendre le problème à résoudre
- Comment obtenir, grâce à la conception orientée objet, une solution robuste, extensible et fiable
- Comment documenter l'analyse et la conception de vos applications à l'aide de UML (*Unified Modeling Language*)

À force de se concentrer sur la syntaxe du langage C++, on risque de perdre de vue l'importance de ces techniques de construction des programmes.

Les modèles

L'objectif d'un modèle est de créer une abstraction significative du monde réel. Cette abstraction doit être plus simple que la réalité, mais aussi la refléter fidèlement afin que le modèle puisse servir à prévoir le comportement des *choses* dans la réalité.

Un globe terrestre posé sur un bureau en est un exemple classique. Le modèle n'est pas la Terre elle-même et il ne risque pas d'être confondu avec la chose réelle. Cependant l'un est une représentation suffisamment fidèle de l'autre pour nous permettre l'apprentissage de la Terre en étudiant le globe.

Il y a, bien sûr, des simplifications importantes : il ne pleut jamais sur le globe de mon fils et il n'est pas sujet aux marées, aux tremblements de terre, etc. Cependant, je peux l'utiliser pour prévoir le temps qu'il me faudra pour me rendre de chez moi à New York.

Un modèle qui ne serait pas plus simple que ce qu'il représente ne serait pas d'une grande utilité.

La conception orientée objet impose la construction de modèles efficaces, elle met en jeu deux éléments importants : un langage et un processus de modélisation.

Conception : le langage de modélisation

Le *langage de modélisation* est l'aspect le moins important de l'analyse et la conception orientée objet, mais c'est malheureusement à lui que s'intéressent principalement la plupart des gens. Un langage de modélisation n'est rien de plus qu'une convention permettant de définir un modèle sur un autre support – papier ou système informatique – et dans un autre format comme les graphiques, le texte ou les symboles. Rien ne nous empêche, par exemple, de représenter nos classes sous la forme de triangles et la relation d'héritage comme une ligne pointillée. Vous pourriez alors modéliser un géranium comme sur la Figure 11.1.

Cette figure montre qu'un géranium est une sorte particulière de fleur. Tant que nous sommes d'accord sur cette représentation du concept d'héritage (généralisation/spécialisation), nous nous comprendrons parfaitement. Au cours du temps, nous voudrons sûrement modéliser de nombreuses relations complexes et nous développerons donc nos propres conventions et règles pour nos diagrammes.

Il reste bien entendu à faire connaître ces conventions à toutes les personnes qui travaillent avec nous et à les apprendre aux nouveaux venus. Si nous collaborons avec d'autres sociétés qui utilisent leurs propres conventions, nous devrons alors mettre en place une convention commune de manière à éviter les malentendus inévitables.

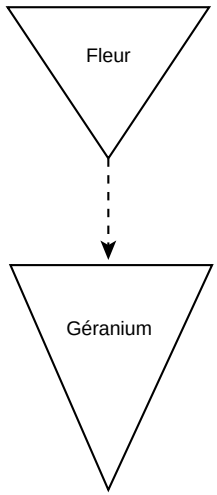


Figure 11.1
Généralisation/ spécialisation.

Il serait encore plus pratique d'avoir recours à un langage commun défini une fois pour toutes. Le *langage universel* en matière d'analyse et de conception logicielle s'appelle UML (*Unified Modeling Language*). Son rôle est de répondre à des questions comme "Comment représenter une relation d'héritage ?". En UML, le géranium de notre exemple aurait ainsi été représenté par la Figure 11.2.

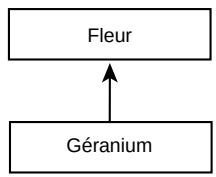


Figure 11.2
Représentation UML de la spécialisation.

En UML, les classes sont représentées par des rectangles et l'héritage sous la forme d'une flèche partant de la classe la plus spécialisée vers la plus générale. Le sens de cette flèche peut sembler peu intuitif, mais il n'a pas vraiment d'importance ; l'essentiel, après avoir appris la représentation, consiste à utiliser une convention commune pour communiquer.

Les détails de UML sont assez simples. Les représentations schématiques sont généralement assez faciles à comprendre et à appliquer et vous les apprendrez à mesure que nous les présenterons. Bien qu'il soit possible d'écrire un livre entièrement consacré à UML, la vérité est que l'on n'en utilise la plupart du temps qu'un petit sous-ensemble, qui s'apprend rapidement.

Conception : le processus

Le *processus* de l'analyse et de la conception orientée objet est bien plus complexe et plus important que le langage de modélisation. Il est donc assez amusant que l'on en entende beaucoup moins parler. En effet, le débat sur les langages de modélisation est plus ou moins terminé – l'industrie a choisi UML comme standard, alors que le débat sur le processus fait toujours rage.

Une *méthode* est un langage et un processus de modélisation. On la désigne souvent par le terme de "méthodologie", bien que ce soit une erreur puisque ce mot désigne l'étude des méthodes.

Il existe plusieurs méthodes, parmi lesquelles la méthode de Grady Booch, celle de Ivar Jacobson et la méthode OMT de James Rumbaugh (*Object Modeling Technolog*t). Ces trois hommes ont concocté *Objectory*, désormais appelé *Rational Unified Process*, qui est à la fois une méthode et un produit commercial distribué par Rational Software, Inc. Lorsqu'ils travaillaient pour cette division d'IBM, ils étaient affectueusement appelés les *Three Amigos*.

Ce chapitre respecte en gros leur processus, sans le suivre aveuglément. D'autres méthodes sont intéressantes à différents points de vue, l'association des unes et des autres permettant d'obtenir une schématique assez efficace. Tout le monde n'adopte pas cette approche ; il est donc conseillé de vous documenter sur la pratique de l'ingénierie logicielle pour vous faire votre propre idée.

Le processus de la conception peut être *itératif*, ce qui signifie qu'au cours du développement du logiciel, la totalité du processus peut être répétée à mesure que vous ajoutez des améliorations découlant d'une meilleure compréhension des besoins. La conception guide l'implémentation, mais les détails absents de l'implémentation réalimentent la conception. Il est important de ne pas tenter, dès le départ, de développer un projet important de façon rectiligne et séquentielle, mais plutôt de boucler dans différentes parties du projet en améliorant constamment sa conception et en affinant son implémentation.

Développement itératif ou développement en cascade ?

Le développement itératif se distingue du développement en cascade, dans lequel la sortie d'une étape devient l'entrée de la suivante et où il est impossible de revenir en arrière (voir Figure 11.3).

Dans un processus en cascade, les besoins sont détaillés et confirmés par le client, ils sont alors transmis au concepteur et gravés sans le marbre. Le concepteur fait son travail puis passe sa conception au programmeur qui l'implémente. Le programmeur écrit le code qui est testé par le Service Qualité, puis livré au client. Belle théorie qui peut se révéler catastrophique en pratique.

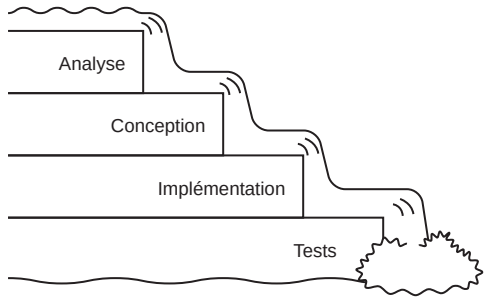


Figure 11.3
La méthode en cascade.

Processus du développement itératif

Dans un développement itératif, on part d'un concept (une idée de ce que l'on veut construire). Cette vision s'étend et évolue à mesure que l'on examine les détails.

Dès que nous avons une idée des besoins, la conception démarre tout en sachant que les questions qui vont se poser entraineront des modifications qui auront probablement un effet sur les besoins. Tout en travaillant sur la conception, on commence également le prototypage puis l'implémentation du produit. Les problèmes survenant pendant le développement ont une répercussion sur la conception et peuvent même influencer notre compréhension des besoins. Nous ne concevons et implémentons que des parties du produit complet, en rebouclant continuellement dans les phases de conception et d'implémentation.

Bien que les étapes du processus se répètent, il est pratiquement impossible de les décrire de manière cyclique ; c'est la raison pour laquelle la liste qui suit les décrit de manière séquentielle.

Étapes du processus de développement itératif :

1. **Conceptualisation.**

   La conceptualisation est la "vision". C'est une simple phrase qui décrit la grande idée.

2. **Analyse.**

   L'analyse est le processus de compréhension des besoins.

3. **Conception.**

   La conception est la phase de création du modèle de vos classes, à partir duquel vous produirez votre code.

4. **Implémentation.**

   L'implémentation est la transcription dans le langage choisi (C++, par exemple).

5. **Tests.**

   Les tests consistent à vérifier que le travail a été bien fait.

6. **Livraison.**

   La livraison est la remise au client.



Ce ne sont pas les mêmes étapes que les phases de Rational Unified Process, qui sont les suivantes :

- *Mise en route ;*
- *Élaboration ;*
- *Construction ;*
- *Transition.*

Ni de son workflow, qui est constitué des étapes suivantes :

- *Modélisation d'activité ;*
- *Besoins ;*
- *Analyse et conception ;*
- *Implémentation ;*
- *Tests ;*
- *Déploiement ;*
- *Configuration et gestion des modifications ;*
- *Gestion de projet ;*
- *Environnement.*

Il doit être bien clair que chacune de ces étapes sera répétée de nombreuses fois pendant le développement d'un produit donné. Pour bien comprendre le processus de conception itératif, il est toutefois plus simple de les présenter l'une après l'autre.

Ce processus est simple. Le reste du chapitre ne fait qu'en présenter les détails.

Controverses

Ce qui se passe au cours de chacune des étapes d'un processus de conception itératif, même le nom attribué à ces étapes, fait l'objet de controverses sans fin. En fait, *cela n'a aucune importance*. Les étapes essentielles sont les mêmes dans tous les processus : savoir ce que vous voulez construire, concevoir une solution et l'implémenter.

Le but est de produire un code qui respecte les besoins énoncés, qui soit fiable, extensible et facilement mis à jour, tout en respectant les délais et le budget impartis.

Étape 1 – La conceptualisation : commencer par la vision

Tout bon logiciel commence par une vision, un produit ou un service que quelqu'un pense intéressant de développer. La vision d'un projet est rarement le fait d'un comité.

La toute première phase d'une analyse et d'une conception orientées objet consiste à capturer cette vision sous la forme d'une simple phrase (ou, au plus, d'un court paragraphe), qui deviendra le principe directeur du développement. L'équipe qui sera chargée d'implémenter cette vision pourra s'y référer – et la modifier si nécessaire – à mesure de l'avancement du projet.

Même si l'énoncé de la vision a été produit par un comité du département financier, une seule personne devrait être considérée comme le "visionnaire". Son travail consiste à être le gardien de la lumière sacrée. À mesure que le projet avance, les besoins évoluent ; les contraintes de planification et de livraison peuvent (et devraient) modifier ce que vous tentez d'accomplir à la première itération du programme, mais le visionnaire doit garder un œil sur l'idée essentielle pour s'assurer que ce qui est produit reflète très fidèlement la vision initiale. C'est ce dévouement sans faille, passionné, qui permettra d'amener le projet à ses fins. Si cette vision est perdue, le produit l'est aussi.

La phase de conceptualisation, dans laquelle s'articule la vision, est très courte. Elle ne doit pas dépasser le temps du flash révélateur et de sa retranscription. Dans d'autres projets, la vision exige une phase de détermination de l'envergure plus complexe, où il faut parvenir à un accord sur les composants, entre les personnes impliquées. Dans ce cas, le succès du projet sera fonction des éléments retenus ou refusés.

Étape 2 – L'analyse : collecter les besoins

Certaines sociétés confondent la vision et les besoins. Une bonne vision est nécessaire, mais pas suffisante. Pour passer à la conception, il faut comprendre l'utilisation qui sera faite du produit et le comportement qu'il devra avoir. L'objectif de l'analyse est d'articuler et de collecter ces besoins. Son résultat est un document des besoins, dont la première partie est l'analyse des case d'utilisation (*use cases*).

Les cas d'utilisation

Les *use cases* , ou *cas d'utilisation*, sont tout simplement une description de haut niveau de la façon dont le produit sera utilisé. Ils orientent non seulement l'analyse, mais également la conception en vous aidant à déterminer les classes et sont particulièrement importants pour les tests du produit.

L'élaboration d'un jeu complet de cas d'utilisation peut représenter le travail le plus important de l'analyse. Les experts du domaine sont les plus qualifiés pour vous aider dans cette tâche car ce sont eux qui ont le plus d'informations sur les besoins que vous tentez de cerner.

Les cas d'utilisation ne tiennent pas compte des détails de l'interface utilisateur, pas plus que des rouages internes du système que vous êtes en train de construire. Ils devraient plutôt se consacrer aux interactions qui doivent survenir et aux personnes et aux systèmes (les *acteurs*) qui devront collaborer pour produire le résultat attendu. .

Voici quelques définitions pour résumer cette partie :

- **Cas d'utilisation.** Une description de la façon dont le logiciel sera utilisé.
- **Experts du domaine.** Comme le nom l'indique, ce sont les experts du domaine auquel le produit est destiné.
- **Acteur.** Tout utilisateur ou système interagissant avec le système que vous développez.

Un cas d'utilisation est une description de l'interaction entre un acteur et le système lui-même. Pour les besoins de l'analyse des cas d'utilisation, le système est considéré comme une "boîte noire". Un acteur "envoie un message au système", et une réaction se produit : des informations sont renvoyées, l'engin spatial change de trajectoire, etc.

Les cas d'utilisation ne suffisent pas à connaître tous les besoins, mais ils sont essentiels et reçoivent souvent la plus grande attention. On peut également inclure les règles d'entreprise, les données et les besoins techniques en matière de performances, de sécurité, etc.

Identifier les acteurs

Les acteurs ne sont pas nécessairement des personnes, les systèmes qui interagissent avec le système que vous construisez le sont aussi. Si, par exemple, vous construisez un guichet automatique de banque, le client et l'employé de banque sont tous les deux acteurs – de même que tout système impliqué, comme un système de prêt ou de suivi des découverts. Les caractéristiques essentielles d'un acteur sont :

- d'être externes au système ;
- d'interagir avec le système.



Le démarrage est souvent la partie la plus difficile de l'analyse des cas d'utilisation. Le meilleur point de départ peut être une séance de brainstorming. Notez la liste des personnes et des systèmes qui interagiront avec votre nouveau système. N'oubliez pas que Personnes signifie en réalité rôles – le guichetier, le directeur, le client, etc. Une même personne peut avoir plusieurs rôles.

Dans notre exemple de guichet automatique, nous pourrions avoir les rôles suivants :

- le client ;
- le personnel de la banque ;
- un système back-office ;
- la personne qui alimente le guichet automatique.

Cette liste suffit pour commencer la génération de nos cas d'utilisation.

Déterminer les premiers cas d'utilisation

Dans l'exemple du guichet automatique, commençons par le *client*. Il peut :

- vérifier son solde ;
- déposer de l'argent sur son compte ;
- retirer de l'argent ;
- faire des virements entre ses comptes ;
- ouvrir un compte ;
- fermer un compte.

Doit-on faire la distinction entre "déposer de l'argent sur son compte courant" et "déposer de l'argent sur son compte épargne", ou ces actions doivent-elles être combinées en une seule (comme nous l'avons fait) : "déposer de l'argent sur son compte" ? Cela dépend si la différence est significative dans le domaine en question.

Pour le savoir, vous devez demander si les *mécanismes* (actions entreprises par le client) et les *sorties* (réponse du système) sont différents. La réponse aux deux questions est "non" : le client dépose de l'argent de la même façon sur ces deux comptes et la sortie est quasiment la même ; le guichet répond en incrémentant le solde du compte concerné.

L'acteur et le système se comportant et répondant de façon quasiment identique dans le cas d'un versement sur l'un ou l'autre compte, ces deux cas d'utilisation n'en forment réellement qu'un seul. Vous pourrez plus tard tenter deux variantes pour voir s'il existe une quelconque différence.

En examinant chaque acteur, vous pouvez découvrir des cas d'utilisation supplémentaires en répondant aux questions suivantes :

- Pourquoi l'acteur utilise-t-il ce système ?

  Le client l'utilise pour obtenir du liquide, faire un dépôt ou vérifier son solde.

- Quelle est la sortie que l'acteur veut ou attend ?

  Ajout d'argent sur un compte ou obtention de liquide pour faire un achat.

- Qu'est-ce qui incite l'acteur à utiliser maintenant le système ?

  Il peut avoir reçu de l'argent sur son compte ou avoir l'intention de faire un achat.

- Que doit faire l'acteur pour utiliser le système ?

  S'identifier en insérant une carte dans la machine.

  Ah ! *Nous avons donc besoin d'un cas d'utilisation supplémentaire pour la connexion du client au système*.

- Quelles informations l'acteur doit-il fournir au système ?

  Entrer un code personnel.

  Ah ! *Nous avons donc encore besoin d'une cas d'utilisation supplémentaire pour obtenir et modifier son code personnel.*

- Quelles sont les informations que l'acteur attend du système ?

  Solde du compte, etc.

Vous pouvez aussi vous concentrer sur les attributs des objets du domaine. Le client a un nom, un code et un numéro de compte. Avez-vous des cas d'utilisation pour gérer ces objets ? Un compte a un numéro, un solde et un historique des opérations. Ces éléments figurent-ils dans les cas d'utilisation ?

Après avoir exploré en détail les cas d'utilisation du client, l'étape suivante consiste à développer ceux des autres acteurs. Dans notre exemple :

- Le client vérifie son ou ses soldes.
- Le client dépose de l'argent sur son ou ses comptes.
- Le client retire de l'argent de son ou ses comptes.
- Le client fait des virements.
- Le client ouvre un compte.
- Le client ferme un compte.
- Le client se connecte à son ou ses comptes.
- Le client vérifie ses dernières transactions.
- L'employé de banque se connecte à un compte spécial de gestion.
- L'employé de banque modifie un compte client.
- Un système back-office met un compte client à jour en fonction d'une activité externe.

- Les modifications d'un compte client sont reflétées dans un système back office.
- Le guichet automatique signale qu'il est à court d'argent.
- Le technicien alimente le guichet automatique en argent et en imprimés.

Création du modèle du domaine

Après avoir établi un premier découpage de vos cas d'utilisation, le document des besoins peut être étoffé avec un *modèle du domaine* détaillé. Ce document capture tout ce que vous savez du domaine. Les objets du domaine que vous créez, décrivant tous les objets mentionnés dans vos cas d'utilisation, en font partie. Notre exemple comprend pour l'instant les objets suivants : client, personnel de la banque, systèmes back-office, compte courant, compte épargne, etc.

Pour chacun de ces objets du domaine, nous devons capturer des données essentielles comme le nom de l'objet (client, compte, etc.), si l'objet est un acteur, ses attributs et ses comportements principaux, etc. De nombreux outils de modélisation permettent de capturer ce type d'informations dans des descriptions de "classe". La Figure 11.4 montre comment ces informations sont représentées avec l'outil de modélisation Rational Rose.

Figure 11.4
Rational Rose.



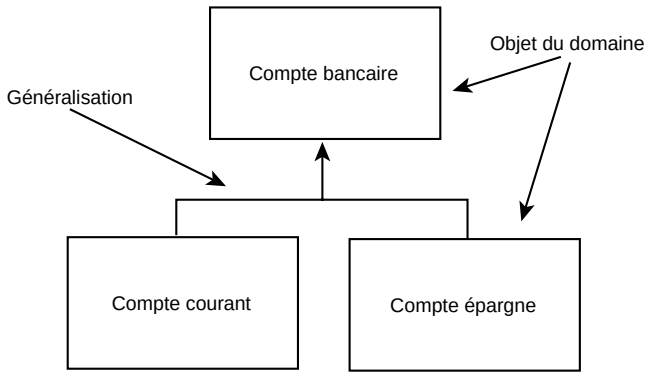
Il est important de réaliser que les objets décrits ne sont *pas* la classe qui sera utilisée dans la conception (même s'il y aura sûrement des classes similaires), mais des classes d'objets du domaine. Il s'agit d'une documentation des besoins de fonctionnement du système, pas d'une documentation de la façon dont celui-ci va répondre à ces besoins.

Les relations entre les objets du domaine de notre exemple peuvent être représentées en UML avec les mêmes conventions que celles que nous utiliserons plus tard pour décrire

les relations entre les classes de la conception. C'est l'un des principaux avantages de UML : vous pouvez utiliser la même représentation à chaque étape du projet.

Par exemple, la Figure 11.5 précise que les vérifications des comptes courant ou épargne sont des spécialisations du concept plus général de compte bancaire.

Figure 11.5
Spécialisation.



Dans la Figure 11.5, les rectangles représentent les différents objets du domaine et les flèches indiquent une généralisation. Le sens des flèches va de la classe *spécialisée* vers la classe *de base* plus générale. Les comptes courant et épargne sont donc tous deux une forme spécialisée de compte bancaire.



Nous montrons ici la relation entre les classes du domaine des besoins. Vous pourrez ultérieurement, au moment de la conception, décider d'avoir un objet `CompteCourant` *et un objet* `CompteBancaire`*, et implémenter cette relation à l'aide de l'héritage, mais ce seront des décisions de conception. Lors de l'analyse, nous nous contentons de documenter notre compréhension de ces objets dans le domaine.*

UML est un langage de modélisation riche permettant de représenter toutes les relations. Les principales, capturées lors de l'analyse, sont les suivantes :

- la généralisation (ou spécialisation) ;
- la composition ;
- l'association.

Généralisation

Elle est souvent assimilée à l'héritage, mais il existe une distinction significative entre les deux. La généralisation décrit la relation ; l'héritage est l'implémentation de la généralisation dans le code. Le corollaire à la généralisation est la spécialisation. Un chat est une forme spécialisée d'animal ; l'animal est un concept généralisé qui réunit un chien ou d'un chat.

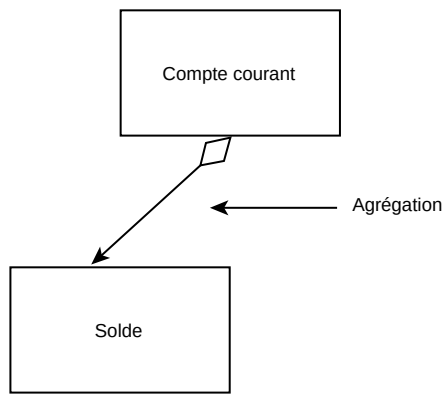
La spécialisation implique que l'objet dérivé *est-un* sous-type de l'objet de base. Un compte courant *est-un* compte bancaire. Cette relation est symétrique : un compte bancaire *généralise* le comportement et les attributs communs d'un compte courant et d'un compte épargne.

Dans la phase d'analyse du domaine, vous devez chercher à capturer ces relations *telles qu'elles existent dans la réalité*.

Composition

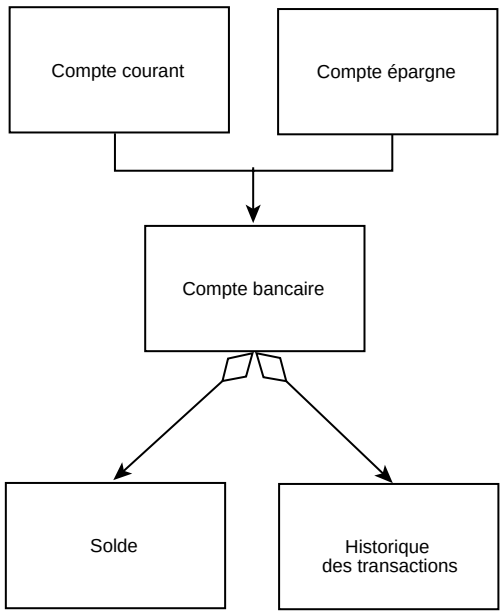
Un objet est souvent composé de plusieurs sous-objets. Un compte courant comprend un solde, un historique des transactions, une identification du client, etc. On dit que le compte courant *a* ces éléments ; la composition modélise la relation *a-un*. En UML, elle est représentée par une flèche avec un losange partant de l'objet contenant vers l'objet contenu, comme le montre la Figure 11.6.

Figure 11.6
Composition.



Le compte courant *a-un* solde. Vous pouvez combiner ces diagrammes pour représenter un ensemble de relations assez complexe. La Figure 11.7 montre qu'un compte courant et un compte épargne sont tous les deux des comptes bancaires et que tous les comptes bancaires ont à la fois un solde et un historique des transactions.

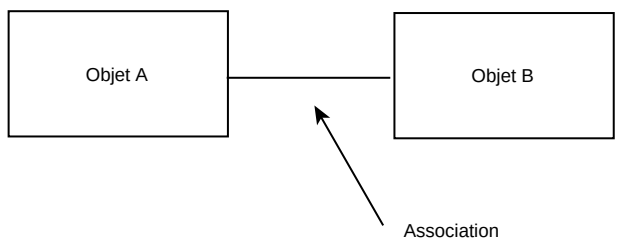
Figure 11.7
Relations entre objets.



Association

La troisième relation fréquemment représentée lors de l'analyse du domaine est une simple association. Une association suggère l'interaction quelconque de deux objets, sans préciser vraiment son objectif. Cette définition sera affinée lors de la phase de conception. Pour l'analyse, elle indique simplement qu'un objet A et un objet B sont liés, mais qu'aucun d'eux ne contient, ni est une spécialisation de l'autre. La notation UML la représente comme une simple ligne entre les objets (voir Figure 11.8).

Figure 11.8
Association.



Établir des scénarios

Avec nos cas d'utilisation préliminaires et les outils permettant de représenter les relations entre les objets du domaine, nous pouvons formaliser les cas et les approfondir.

Chaque cas d'utilisation peut être décomposé en une série de scénarios. Un *scénario* est une description d'un ensemble précis de circonstances qui se distinguent parmi les éléments du cas d'utilisation. Par exemple, le cas "Le client retire de l'argent de son ou ses comptes" peut avoir les scénarios suivants :

- Le client demande un retrait de 300 € de son compte courant, il prend les billets, et le système imprime un reçu.
- Le client demande un retrait de 300 € de son compte courant, mais son solde est de 200 €. Il est avisé que son compte n'est pas suffisamment approvisionné.
- Le client demande un retrait de 300 € de son compte courant, mais il a déjà retiré 100 € aujourd'hui et la limite est de 300 € par jour. Il est informé du problème et doit décider s'il ne veut prélever que 200 €.
- Le client demande un retrait de 300 € de son compte courant, mais le reçu ne peut être imprimé par manque de papier. Il est informé du problème et doit décider s'il veut ou non poursuivre l'opération sans recevoir de reçu.

Ainsi de suite... Chaque scénario explore une variante du cas original et représente parfois des cas exceptionnels (pas assez d'argent sur le compte, plus de papier, etc.). Ils peuvent aussi impliquer des décisions nuancées dans le cas lui-même (le client veut-il transférer de l'argent avant de procéder au retrait ?).

Il est impossible d'explorer tous les scénarios. En réalité, on recherche plutôt ceux qui expriment les besoins du système ou les détails de l'interaction avec l'utilisateur.

Établir des directives

Vous devez créer des directives pour documenter chaque scénario. Celles-ci sont capturées dans vos spécifications des besoins. Généralement, vous devrez vous assurer que chaque scénario comprend :

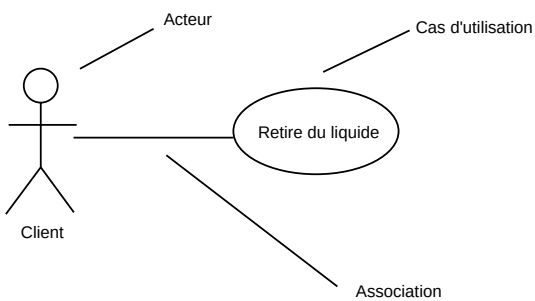
- Des préconditions. Ce qui doit être vrai pour que le scénario commence.
- Des déclencheurs. Les événements qui provoquent le démarrage du scénario.
- Des actions réalisées par l'acteur.
- Des résultats ou modifications provoqués par le système.
- Un retour reçu par l'acteur.
- Des actions répétitives se produisent-elles ? Ce qui leur fait prendre fin.
- Une description du flux logique du scénario.
- Ce qui provoque la fin du scénario.
- Des postconditions. Ce qui doit être vrai lorsque le scénario est terminé.

En outre, vous devrez nommer chaque cas d'utilisation et chaque scénario. Par exemple :

Cas d'utilisation :	Le client demande à retirer du liquide.
Scénario :	Retrait réussi du compte courant.
Préconditions :	Le client est déjà connecté au système.
Déclencheur :	Le client demande un "retrait".
Description :	Le client choisit de retirer du liquide d'un compte courant. Le guichet automatique est approvisionné en argent liquide et en papier pour le reçu, le système fonctionne. Le guichet automatique demande au client le montant à retirer, le client demande 300 €, ce qu'il est autorisé à faire. La machine lui donne 300 € et imprime un reçu, le client prend l'argent et le reçu.
Postconditions :	Le compte du client est débité de 300 €, et le client possède 300 € en liquide.

Ce cas peut être représenté à l'aide du schéma simple de la Figure 11.9.

Figure 11.9
Représentation d'un cas d'utilisation.



La seule information capturée ici est une interaction abstraite entre un acteur (le client) et le système. Le diagramme devient un peu plus utile lorsque l'on montre l'interaction entre les cas d'utilisation. Nous avons écrit *un peu* plus utile parce qu'il n'existe que deux interactions possibles : `<<utilise>>` et `<<étend>>`. La première indique qu'un cas d'utilisation est un surensemble d'un autre. Il n'est, par exemple, pas possible de *retirer de l'argent* sans s'être *identifié*. Cette relation est représentée dans la Figure 11.10.

Cette figure indique que le cas d'utilisation "Retire de l'argent" *utilise* le cas "S'identifie" et que ce dernier fait donc partie du premier.

En fait, le stéréotype `<<étend>>` était censé indiquer une relation conditionnelle ressemblant un peu à l'héritage, mais il y a tant de confusion entre `<<utilise>>` et `<<étend>>` dans la modélisation objet que de nombreux développeurs ont simplement mis de côté `<<étend>>` en estimant que sa signification n'était pas suffisamment bien comprise.

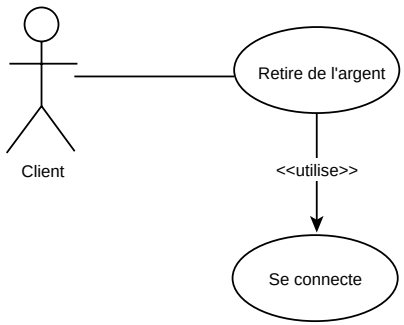


Figure 11.10
Le stéréotype <<utilise>>.

Personnellement, nous utilisons `<<utilise>>` lorsque nous pourrions simplement copier et coller sur place le cas d'utilisation existant et `<<étend>>` lorsque l'on *utilise* le cas d'utilisation sous certaines conditions.

Diagrammes d'interaction

Bien que le diagramme du cas d'utilisation ait, en lui-même, un intérêt limité, vous pouvez associer des diagrammes aux cas d'utilisation pour améliorer la documentation et la compréhension des interactions. Nous savons, par exemple, que le scénario Retirer de l'argent représente les interactions des objets suivants du domaine : client, compte courant et interface utilisateur. Vous pouvez donc documenter cette interaction avec un diagramme d'interaction (également appelé diagramme de collaboration) comme celui de la Figure 11.11.

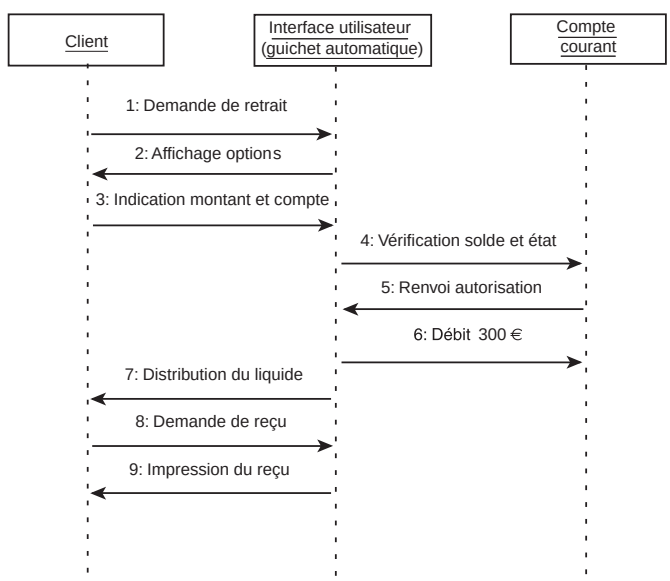


Figure 11.11
Diagramme d'interaction UML.

Le diagramme d'interaction de la Figure 11.11 capture les détails du scénario qui pourraient ne pas apparaître clairement à la lecture du texte. Les objets qui interagissent sont des objets du *domaine*, et l'ensemble "guichet automatique/Interface utilisateur" est traité comme un unique objet, avec seulement le compte bancaire demandé en détail.

Cet exemple de guichet assez simple ne montre qu'un ensemble d'interactions, mais détailler leurs spécificités est un outil puissant, permettant de comprendre à la fois le problème du domaine et les besoins du nouveau système.

Créer des paquetages

Les cas d'utilisation peuvent être regroupés en packages avec UML.

Un *paquetage* ressemble à un répertoire ou à un dossier : c'est une collection d'objets de modélisation (classes, acteurs, etc.). Pour gérer la complexité des cas d'utilisation, vous pouvez les regrouper en paquetages en fonction de critères correspondant à votre problème. Les cas d'utilisation peuvent ainsi être agrégés par type de compte (compte courant ou compte épargne), par crédit ou débit, par type de client ou toute autre caractéristique qui vous semble judicieuse. En outre, un même cas d'utilisation peut apparaître dans des paquetages différents, ce qui offre une grande souplesse.

Analyse de l'application

Outre les cas d'utilisation, la spécification de besoins doit capturer les suppositions du client et toutes les contraintes ou besoins concernant le matériel, les systèmes d'exploitation, la sécurité, les performances, etc. Ces besoins sont les prérequis *particuliers* du client – vous pourriez les découvrir au moment de la conception et de l'implémentation, mais votre client a décidé pour vous.

Les besoins de l'application (quelquefois appelés "besoins techniques") sont souvent dictés par la nécessité de communiquer avec des systèmes existants. La compréhension de ce que font les systèmes existants et de la façon dont ils le font est donc une composante essentielle de l'analyse.

Dans l'idéal, vous analysez le problème, concevez la solution, puis choisissez les plates-formes et systèmes d'exploitation qui sont les mieux adaptés au besoin. Cependant, en pratique, c'est le client qui décide en fonction de son système existant et c'est à vous de vous adapter.

Analyse des systèmes

Certains logiciels sont autonomes et n'interagissent qu'avec le client final. Le plus souvent, vous devrez réaliser une interface avec un système existant. L'*analyse du système* est le processus consistant à recueillir tous les détails concernant le système avec lequel

vous allez interagir. Votre système sera-t-il un serveur fournissant des services au système existant ou sera-t-il un client ? Devrez-vous prévoir une interface nouvelle entre les systèmes ou en adapter une existante ? Les autres systèmes sont-ils stables ou devrez-vous continuellement tenter d'atteindre une cible mouvante ?

Vous devez avoir la réponse à toutes ces questions et à bien d'autres lors de la phase d'analyse, avant de démarrer votre conception. En outre, vous devez essayer de capturer les contraintes et les limites implicites des interactions avec les autres systèmes. Ralentiront-ils la réactivité du vôtre ? Imposeront-ils une charge importante au niveau ressources et temps de traitement ?

Documents de planning

Une fois que vous savez ce que doit faire votre système et comment il doit se comporter, il est temps d'établir une prévision en ce qui concerne le budget et les délais de réalisation. Le plus souvent, c'est le client qui définit la date butoir : "Vous avez 18 mois." Idéalement, vous évaluez le temps et le budget nécessaires à l'implémentation de votre solution, mais, en pratique, vous devez le plus souvent vous arranger pour réaliser le travail dans les délais et avec le budget impartis.

Partez du principe que :

- Si on vous octroie une fourchette, la limite supérieure est probablement optimiste.
- Toute réalisation prendra plus de temps que prévu.

Vous devez donc impérativement établir des priorités afin de commencer par ce qui est le plus important. *N'espérez pas avoir le temps de finir*, c'est aussi simple que ça. Lorsque vous aurez atteint la limite du temps imparti, il faut que ce que vous avez fait fonctionne et puisse représenter une première version satisfaisante. Si vous construisez un pont et que le temps vous manque, vous devez pouvoir y faire circuler au moins une bicyclette. C'est toujours mieux que d'avoir un beau pont bien fignolé, qui s'arrête à la moitié du chemin.

Un autre point essentiel concernant les prévisions est qu'elles sont généralement fausses. À ce stade du processus, il est pratiquement impossible de faire une estimation fiable de la durée du projet. Lorsque vous disposerez des spécifications des besoins, vous pourrez avoir une idée plus précise et établir une estimation en ajoutant au moins 20 à 25 % de marge de sécurité – que vous pourrez réduire à mesure que vous progresserez et que vous en saurez plus.

Visualisation

La dernière pièce de la spécification des besoins est la visualisation, qui est un joli nom pour désigner les diagrammes, les images, les captures d'écrans, les prototypes et toutes

les autres représentations visuelles qui vous permettent d'imaginer et de concevoir l'interface graphique de votre produit.

Pour les très gros projets, vous pouvez développer un prototype complet pour vous aider (ainsi que vos clients) à comprendre comment se comportera le système. Dans certaines équipes, ce prototype devient même la spécification des besoins ; le "vrai" système est alors conçu pour implémenter les fonctionnalités présentées dans le prototype.

Artefacts

À la fin de chaque phase d'analyse et de conception, vous allez créer une série de documents ou (souvent appelés "artefacts"). Le Tableau 11.1 décrit certains artefacts de la phase d'analyse. Ces documents sont utilisés par plusieurs groupes. Le client s'en servira pour vérifier que vous comprenez ses besoins, l'utilisateur final pour vous fournir un retour d'informations et des conseils, l'équipe projet pour concevoir et implémenter le code. La plupart de ces documents fournit également des informations cruciales pour la documentation et indique au service qualité comment devrait se comporter le système.

Tableau 11.1 : Documents créés pendant la phase d'analyse du projet

Document	*Description*
Rapport de cas d'utilisation	Rapport détaillant les cas d'utilisation, scénarios, stéréotypes, préconditions, postconditions et documents visuels
Analyse du domaine	Document et diagrammes décrivant les relations entre les objets du domaine
Diagrammes d'analyse de collaboration	Diagrammes de collaboration décrivant les interactions entre les objets du domaine
Diagrammes d'analyse de l'activité	Diagrammes d'activité décrivant les interactions entre les objets du domaine
Analyse des systèmes	Rapport et diagrammes décrivant les systèmes matériels et de bas niveau sur lesquels sera construit le projet
Analyse de l'application	Rapport et diagrammes décrivant les besoins spécifiques du client pour ce projet particulier
Rapport des contraintes opérationnelles	Rapport décrivant les caractéristiques et contraintes de performance imposées par ce client
Budget et planning	Rapport avec diagrammes et graphiques donnant les prévisions en matière de planning, de repères et de coûts

Étape 3 – La conception

L'analyse met l'accent sur le domaine du problème, alors que l'étape suivante, celle de la conception, se concentre sur la création de la solution, c'est-à-dire la transformation de notre compréhension des besoins en un modèle pouvant être implémenté par un programme. Le résultat de ce processus est la production d'un document de conception.

Ce document peut être divisé en deux sections : les mécanismes d'architecture et la conception des classes. Cette dernière est à son tour divisée en conception statique (détail des classes, de leurs relations et de leurs caractéristiques) et conception dynamique (détail des interactions entre classes).

La section mécanismes d'architecture fournit des détails sur la façon dont vous implémenterez la persistance des objets, les conflits d'accès, un système objet distribué, etc. Le reste de ce chapitre se concentre sur l'aspect conception de classe du document de conception. L'implémentation des différents mécanismes d'architecture sera présentée dans d'autres chapitres.

Qu'est-ce que les classes ?

Vous savez maintenant créer des classes. Une méthode de conception formelle exige de séparer le concept de classe C++ de celui de classe de conception, bien qu'ils soient intimement liés. La classe C++ que vous écrivez est la mise en œuvre de la classe que vous avez conçue. Chaque classe de votre conception correspondra à une classe dans votre code, mais elles ne doivent pas être confondues : il est possible d'implémenter vos classes de conception dans un autre langage, auquel cas il faudra modifier la *syntaxe* des définitions de classes.

Cela dit, la plupart du temps nous parlons de ces classes sans faire la distinction car les différences sont purement abstraites. Si vous dites que, dans votre modèle, votre classe `Chat` aura une méthode `Miauler()`, cela signifie que vous écrirer aussi une méthode `Miauler()` dans votre classe C++.

Vous capturez les classes du modèle dans des diagrammes UML et vous capturez les classes C++ de l'implémentation dans le code qui est compilé. La distinction est subtile, mais significative.

La plus grande difficulté, pour bien des débutants, consiste à déterminer l'ensemble initial des classes et de comprendre ce qui fait une classe bien conçue. Une technique très simple suggère d'écrire les scénarios des cas d'utilisation puis de créer une classe pour chaque nom. Considérez, par exemple, le scénario suivant :

> Le **client** choisit de retirer du **liquide** de son **compte courant**. Le **compte** est suffisamment approvisionné, le **guichet automatique** est alimenté en liquide et en **reçus** et

le **réseau** est opérationnel. Le guichet automatique demande au client d'indiquer un **montant** pour le **retrait** et le client demande 300 ı ce qui est autorisé. La **machine** distribue 300 € et imprime un reçu, le client prend l'**argent** et le reçu.

Vous pouvez déduire de ce scénario les classes suivantes :

- Client
- Liquide
- CompteCourant
- Compte
- Recus
- GuichetAuto
- Reseau
- Montant
- Retrait
- Machine
- Argent

On peut ensuite regrouper les synonymes pour ne conserver que la liste suivante, puis créer des classes pour chacun de ces noms :

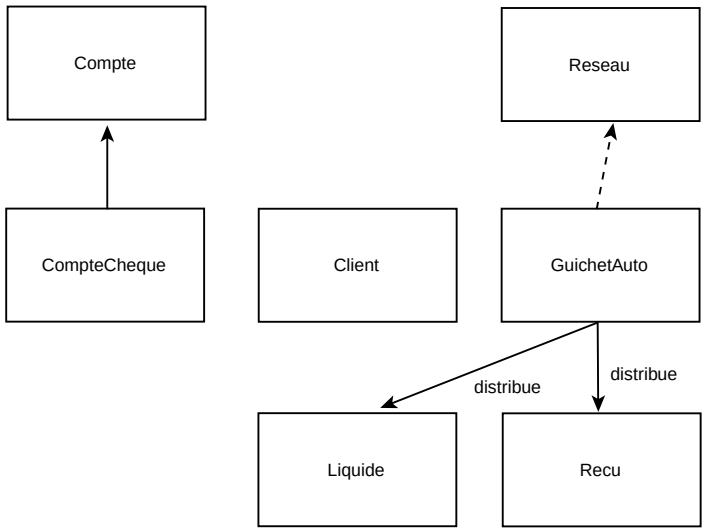
- Client
- Liquide (argent, montant, retrait)
- CompteCourant
- Compte
- Recus
- GuichetAuto (machine)
- Reseau

Ce n'est pas un mauvais début. Vous pouvez ensuite schématiser les relations évidentes entre certaines de ces classes comme sur la Figure 11.12.

Transformations

Cette extraction des noms du scénario dans la précédente section est surtout le début de la transformation des objets provenant de l'analyse du domaine. Ce n'est qu'une première étape. Bien souvent, nombre des objets du domaine auront des *substituts* dans la conception. Un objet est appelé substitut pour distinguer le reçu physique distribué par le guichet de l'objet de la conception, qui n'est qu'une abstraction implémentée dans le programme.

Figure 11.12
Classes préliminaires.



Vous découvrirez probablement que *la plupart* des objets du domaine ont une *représentation* dans la conception – c'est-à-dire qu'il existe une relation un–un entre l'objet du domaine et l'objet de conception. D'autres fois, un même objet du domaine est représenté dans la conception par une série d'objets de conception. Il peut même arriver qu'un ensemble d'objets du domaine soit représenté par un seul objet de conception.

Vous remarquez, dans la Figure 11.12, que l'on a déjà capturé le fait que `CompteCourant` soit une spécialisation de `Compte`. De même, à partir de l'analyse du domaine, nous savons que le `GuichetAuto` distribue à la fois du `liquide` et des `recus` et nous l'avons aussitôt capturé dans la conception.

La relation entre le `Client` et `CompteCourant` est moins évidente. Nous savons qu'une telle relation existe, mais les détails ne sont pas clairs et il est donc préférable de la laisser de côté.

Autres transformations

Souvent, chaque acteur possède une classe. Après avoir transformé les objets du domaine, vous pouvez commencer à rechercher d'autres objets utiles au moment de la conception, à commencer par l'interface entre votre nouveau système et tous les systèmes existants – cela devrait être encapsulé dans une classe de l'interface. Prenez garde toutefois lorsque vous traitez des bases de données et autres supports de stockage externes. Il est généralement préférable de donner la responsabilité à chaque classe de gérer sa propre "persistance" (son stockage et sa récupération entre les sessions utilisateurs). Ces classes de conception peuvent bien entendu utiliser des classes communes pour accéder aux fichiers

ou aux bases de données mais, le plus souvent, ce sera le fournisseur du système d'exploitation ou de la base de données qui vous les procurera.

Ces classes d'interface permettent d'encapsuler les interactions de votre système avec l'autre système et protègent donc votre code des modifications dans ce dernier. Elles vous permettent également de modifier votre propre conception ou de satisfaire les changements dans la conception des autres systèmes, sans altérer le reste du code. Tant que les deux systèmes continuent de respecter l'interface convenue, ils peuvent évoluer indépendamment l'un de l'autre.

Manipulation des données

De la même façon, vous devrez peut-être créer des classes pour la manipulation des données. Si vous devez transmettre des données d'un format vers un autre (de degrés Fahrenheit en Celsius par exemple), vous encapsulerez ces manipulations dans une classe spéciale. Cette technique peut servir à chaque fois que vous devez transformer des données dans les formats exigés par les autres systèmes ou pour les transmettre sur Internet – en résumé, à chaque fois que vous devez manipuler des données dans un format précis, il faut encapsuler le protocole dans une classe de manipulation.

Vues et rapports

Chaque "vue" ou "état" produit par votre système (ou, si vous produisez de nombreux états, chaque ensemble d'états) est un candidat pour une classe. Les règles conditionnant la production de l'état – comment les informations sont rassemblées et affichées – peuvent être encapsulées dans une classe `Vue`.

Périphériques

Si votre système utilise des périphériques (imprimantes, appareils photo, modems, scanners, etc.), les caractéristiques de leurs protocoles doivent être encapsulées dans une classe. Ici encore, en créant des classes pour l'interface vers le périphérique, vous pouvez en brancher de nouveaux avec de nouveaux protocoles sans toucher au reste du code ; il suffit de créer une nouvelle classe d'interface avec la même interface (ou dérivant de celle-ci).

Construction du modèle statique

Votre ensemble préliminaire de classes étant établi, vous pouvez commencer à modéliser leurs relations et interactions. Nous décrivons ici d'abord le modèle statique, puis le modèle dynamique, mais, dans le processus réel de conception, vous passerez de l'un à l'autre pour en compléter les détails – en ajoutant de nouvelles classes et en les ébauchant au fur et à mesure de votre progression.

Les trois points essentiels du modèle statique sont : les responsabilités, les attributs et les relations. Le principal – celui, qu'il faut considérer en premier lieu – est l'ensemble des responsabilités de chaque classe. Le principe essentiel est : *chaque classe doit être responsable d'une seule chose*.

Ceci ne signifie pas qu'une classe ne possède qu'une méthode, loin de là. De nombreuses classes en ont des douzaines, mais toutes ces méthodes doivent être cohérentes et cohésives ; elles doivent être mutuellement apparentées et contribuer à la capacité de la classe à exercer un unique domaine de responsabilité.

Chaque objet doit être une instance d'une classe bien définie chargée d'une responsabilité particulière. Les classes délèguent généralement les responsabilités annexes à d'autres classes connexes, ce qui facilite la maintenance d'un programme.

Pour ébaucher les responsabilités de vos classes, vous pouvez réunir un groupe de trois à six personnes, dont un informaticien ayant une expérience de l'analyse et de la conception orientée objet, un ou deux experts du domaine, et un "meneur" qui gardera un œil sur l'objectif et recentrera la discussion.

Le but est de définir les classes à l'aide de fiches, d'identifier leurs responsabilités et de comprendre leurs interactions.

Prévoyez une série de fiches d'environ 10×15 cm, sur lesquelles vous placerez le nom d'une classe en titre, puis séparez la fiche en deux colonnes : Responsabilités et Collaboration. Commencez par compléter les fiches pour les classes les plus importantes que vous avez identifiées, écrivez au dos une simple phrase pour les définir. Si vous pouvez également savoir quelle est la classe que cette classe spécialise, écrivez simplement "Superclasse:" sous le nom de la classe, suivi du nom de la classe dont elle dérive.

Ne vous attardez pas sur les attributs des classes, ne capturez que ceux qui sont évidents et essentiels. Concentrez-vous sur leurs responsabilités, en mentionnant simplement dans la colonne Collaboration si la classe doit déléguer à une autre.

Si la place vient à manquer sur une fiche, demandez-vous si vous ne déléguez pas trop à cette classe. Les méthodes d'une classe doivent être cohérentes et cohésives et contribuer à l'achèvement de la responsabilité globale de la classe.

À ce stade, ne vous préoccupez pas des relations ni de l'interface, pas plus que de savoir si telle ou telle méthode sera publique ou privée.

Une fois cette série de fiches établie, distribuez-les arbitrairement dans le groupe et reprenez votre scénario précédent :

> Le **client** choisit de retirer du **liquide** de son **compte courant**. Le **compte** est suffisamment approvisionné, le **guichet automatique** est alimenté en liquide et en **reçus**, et le **réseau** est opérationnel. Le guichet automatique demande au client d'indiquer un

montant pour le **retrait**, et le client demande 300 € ce qui est autorisé. La **machine** distribue 300 € et imprime un reçu, le client prend l'**argent** et le reçu.

La suite ressemble à un jeu de rôles où chaque participant se met dans la peau du personnage qui donnerait vie à chaque classe. Par exemple, le participant possédant la fiche `CompteCourant` va dire : "Je dis au client la somme qui est disponible sur le compte. Il me demande 300 €, j'envoie un message à la machine pour lui demander de le faire." Le titulaire de la fiche `GuichetAuto` prend alors la parole : "Je suis le distributeur, je donne 300 € et j'envoie à `CompteCourant` un message pour lui dire de débiter le compte de 300 €. La machine dispose de 300 € de moins. À qui dois-je le dire ? Est-ce que je dois en garder la trace ?, etc.

Cette première approche permet de clarifier les responsabilités de chaque classe et leurs interactions.

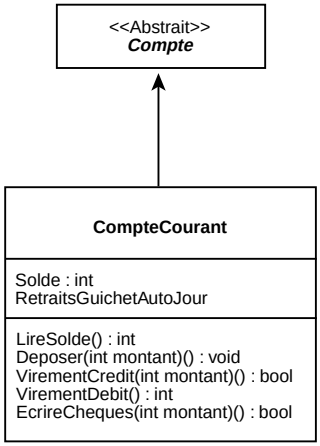
Cette méthode a toutefois ses limites. Elle n'est tout d'abord par utilisable à grande échelle ; pour un projet très complexe, vous serez vite dépassé. Par ailleurs, ce système de fiches ne permet pas d'appréhender les relations interclasses. Bien que les collaborations soient capturées, leur nature n'est pas bien modélisée. Il est difficile de passer des fiches au code et, plus important encore, les fiches sont statiques.

Pour résumer, les fiches sont un bon début, mais vous devrez évoluer vers la notation UML pour pouvoir modéliser un modèle complet et fiable de votre conception. Cette transition *vers* UML n'est pas très difficile, mais elle est irréversible. Vous devez définitivement renoncer aux fiches, car il est tout simplement impossible de conserver les deux modèles synchronisés.

Chaque fiche peut être traduite directement en une classe UML, les responsabilités en méthodes de la classe, et les attributs capturés ajoutés. La définition de classe, inscrite au dos de la fiche, devient la documentation de la classe. La Figure 11.13 montre la relation entre la fiche `CompteCourant` et la classe UML correspondante.

Classe :	`CompteCourant`
SuperClasse :	`Compte`
Responsabilités :	Garder la trace du solde
	Accepter les dépôts et virements à créditer
	Écrire les chèques
	Transférer le liquide demandé
	Mettre à jour le solde du jour des retraits de la machine
Collaborations :	Autres comptes
	Systèmes back-office
	Distributeur de liquide

Figure 11.13
Fiche CompteCourant.



Relations entre les classes

Lorsque les classes sont transcrites en notation UML, vous pouvez commencer à vous préoccuper des relations entre les différentes classes. Les relations principales à modéliser sont les suivantes :

- généralisation ;
- association ;
- agrégation ;
- composition.

La relation de généralisation est implémentée en C++ par le biais de l'héritage public. Du point de vue conception, c'est plus la sémantique qui nous importe (ce qu'implique cette relation) que son mécanisme.

Nous avons vu la relation de généralisation dans la phase d'analyse, mais nous nous concentrons à présent sur les objets dans notre conception plutôt que simplement sur les objets dans le domaine. Nous devons "factoriser" les fonctionnalités communes des classes connexes dans les classes de base pour encapsuler les responsabilités partagées.

Cette "factorisation" consiste à déplacer la fonctionnalité des classes spécialisées vers la classe plus générale. Si vous remarquez que vos comptes courant et épargne ont tous les deux besoin de méthodes pour transférer de l'argent dans et dehors du compte, vous allez déplacer la méthode `TransférerFonds()` dans la classe de base `Compte`. Cette pratique permet d'accentuer le caractère polymorphe de votre conception.

Une fonctionnalité de C++, non disponible dans Java, est le concept d'*héritage multiple* (Java dispose d'une fonctionnalité similaire, bien que limitée, les *interfaces multiples*).

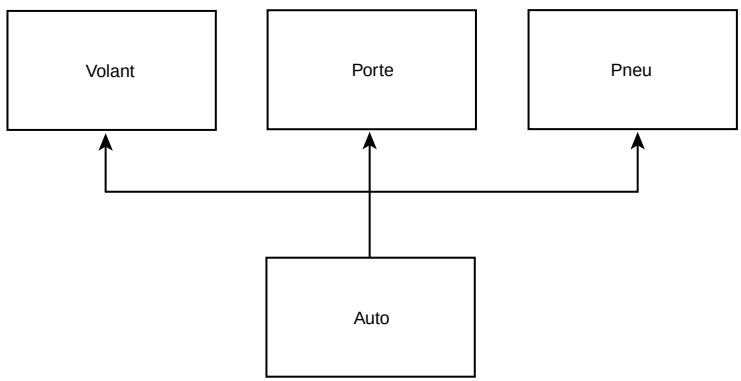
L'héritage multiple permet à une classe d'hériter de membres et méthodes de plusieurs classes de base.

L'héritage multiple doit être utilisé judicieusement, car il peut compliquer à la fois la conception et l'implémentation. Pour cette raison, de nombreux problèmes autrefois résolus par l'héritage multiple le sont aujourd'hui par l'agrégation. Ceci dit, l'héritage multiple est un outil puissant si votre conception nécessite qu'une unique classe spécialise le comportement de deux ou plusieurs autres classes.

Héritage multiple ou composition

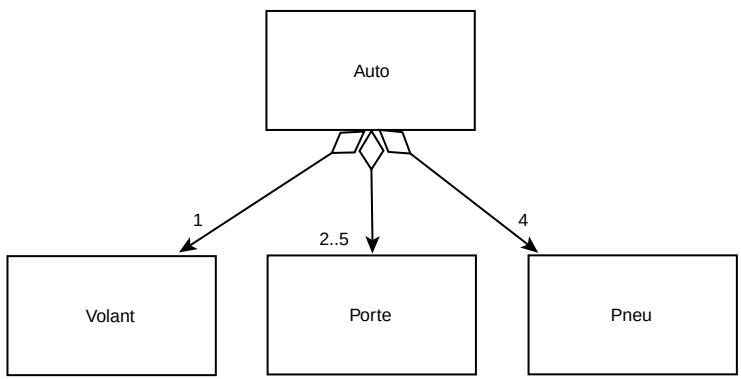
Un objet est-il la somme de ses parties ? Est-il logique de modéliser un objet `Auto` en tant que spécialisation de `Volant, Porte`, et `Pneu`, comme sur la Figure 11.14 ?

Figure 11.14
Faux héritage.



Petit rappel : l'héritage public doit toujours modéliser la généralisation. On dit que l'héritage doit modéliser une relation *est-un*. Pour modéliser une relation *a-un* – par exemple, une auto *a-un* volant – il faut utiliser l'agrégation (voir Figure 11.15).

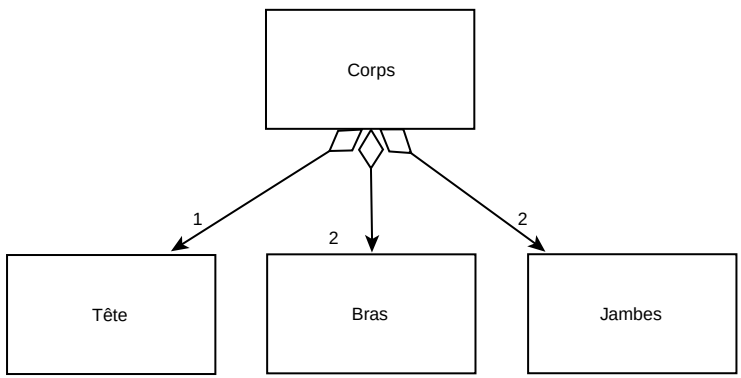
Figure 11.15
Agrégation.



Le diagramme de la Figure 11.15 indique qu'une auto *a-un* volant, 4 roues et 2 à 5 portes, ce qui est un meilleur modèle de la relation entre une auto et ses éléments. Remarquez que le losange est évidé puisqu'il s'agit d'une agrégation et non d'une composition. Cette dernière implique le contrôle de la durée de vie de l'objet. Bien qu'une auto *ait* des pneus et une porte, ceux-ci existent avant de faire partie de l'auto et existent encore même s'ils n'en font plus partie.

La Figure 11.16 modélise la composition. Ce modèle indique que le corps n'est pas seulement une agrégation d'une tête, de deux bras et deux jambes, mais que ces objets sont créés en même temps que le corps et disparaissent en même temps que lui. Leur existence dépend de celle du corps et leur durée de vie est inextricablement liée.

Figure 11.16
Composition.



Discriminateurs

Comment procéder pour définir les classes reflétant les différents modèles d'un fabricant d'automobiles ? Supposons que vous ayez dans la Société UniversAuto les modèles suivants : la Pluton (petite avec un petit moteur), la Vénus (berline 4 portes avec un moteur moyen), la Mars (coupé sport avec le moteur le plus puissant), la Jupiter (mini-fourgon avec le même moteur que le coupé, mais conçu pour avoir plus de couple afin de gérer le poids plus élevé de ce modèle), et la Globe (un 4×4).

Vous pouvez commencer par créer les sous-types reflétant les différents modèles, puis créer des instances de chaque à la sortie des chaînes de montage (voir Figure 11.17).

Qu'est-ce qui différencie ces modèles ? La taille du moteur, le type de carrosserie et les performances. Ces caractéristiques peuvent être combinées pour créer les différents modèles. En UML cela se traduit par le stéréotype de *discrimination* (voir Figure 11.18).

Le diagramme de la Figure 11.18 indique que les classes peuvent être dérivées de `Auto` en combinant trois attributs. Vous pouvez avoir un 4×4 puissant et sport, une berline familiale peu puissante, etc.

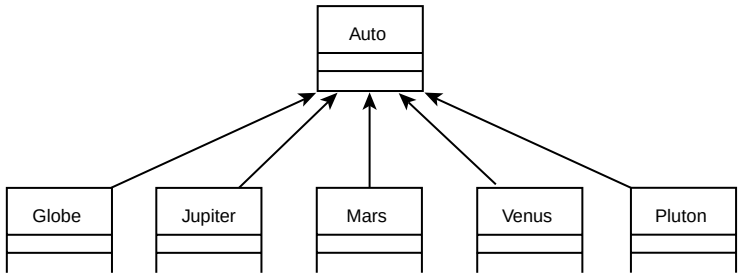


Figure 11.17
Modélisation des sous-types.

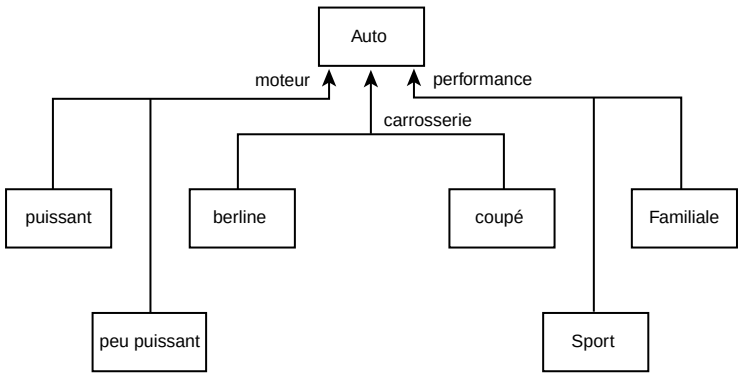


Figure 11.18
Modélisation du discriminateur.

Chaque attribut peut être implémenté à l'aide d'une simple énumération dans le code. Par exemple :

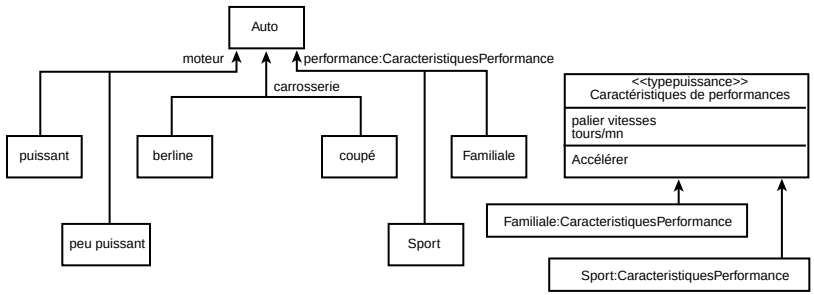
```
enum TypeCarrosserie = { berline, coupe, miniFourgon, quatre4 };
```

Une seule valeur peut s'avérer insuffisante pour modéliser un discriminateur particulier : les performances, par exemple, forment une caractéristique assez complexe. Le discriminateur peut alors être modélisé comme une classe et la discrimination encapsulée dans une instance de ce type.

Les caractéristiques de performance seront donc modélisées dans un type *performance*, qui contiendra toutes les informations techniques du moteur. Le stéréotype UML pour une classe qui encapsule un discriminateur et qui peut être utilisé pour créer des *instances* d'une classe (`Auto`) de différents types est `<<typepuissance>>`. Ici la classe `Performance` est un `typepuissance` pour `Auto`. Vous instanciez un objet `Performance` en même temps qu'un `Auto`, et vous associez un objet `Performance` donné à un objet `Auto` donné, comme sur la Figure 11.19.

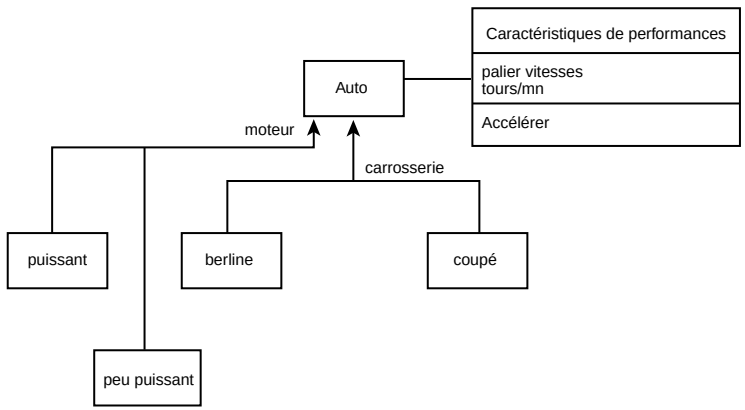
Les `typepuissance` permettent de créer des types *logiques* sans utiliser l'héritage et de gérer un ensemble important et complexe de types.

Figure 11.19
Un discriminateur en tant que typepuissance.



En C++, ils sont généralement implémentés à l'aide de pointeurs. Ici, la classe `Auto` contient un pointeur sur une instance de la classe `CaracteristiquesPerformance` (voir Figure 11.20).

Figure 11.20
La relation entre un objet `Auto` et son `typepuissance`.





La création de nouveaux types au moment de l'exécution avec cette méthode peut réduire les avantages du typage fort en C++, dans lequel le compilateur peut faire appliquer la justesse des relations interclasses. À utiliser avec précaution.

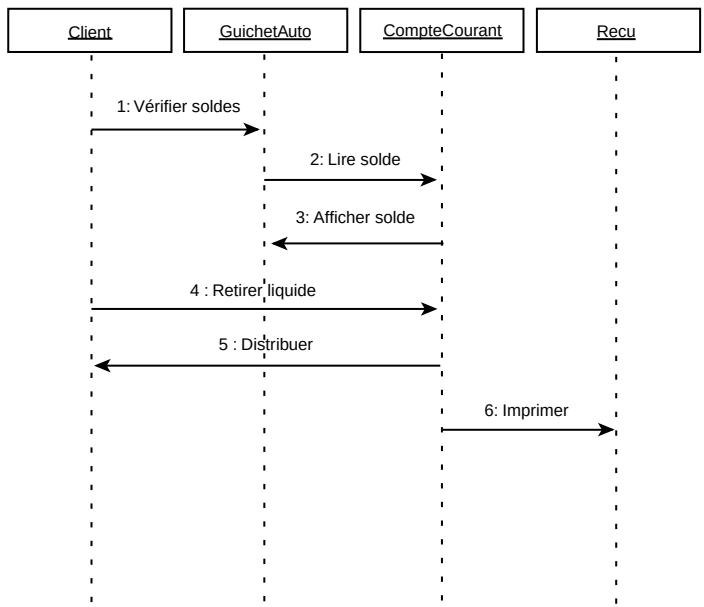
```
Class Car : public Vehicule
{
public:
    Auto();
    ~Auto();
    // autres méthodes publiques
private:
    CaracteristiquesPerformance * pPerformance;
};
```

Enfin, les `typepuissance` permettent de créer de nouveaux *types* (pas seulement des instances) au moment de l'exécution. Chaque type logique étant différencié seulement par les attributs du `typepuissance` associé, ils peuvent être les paramètres du constructeur de `typepuissance` et vous pouvez créer de nouveaux *types* d'auto à la volée au moment de l'exécution. En passant des tailles de moteur ou des paliers de vitesse différents au `typepuissance`, vous pouvez créer de nouvelles caractéristiques de performance. En affectant ces caractéristiques aux différentes autos, vous pouvez agrandir le jeu des types de voitures *au moment de l'exécution*.

Modèle dynamique

Outre les relations entre classes, il est important de modéliser leurs interactions. Par exemple, les classes `CompteCourant`, `GuichetAuto` et `Recu` peuvent interagir avec le `Client` par l'intermédiaire du cas d'utilisation "Retirer liquide" (voir Figure 11.21).

Figure 11.21
Diagramme de séquence.

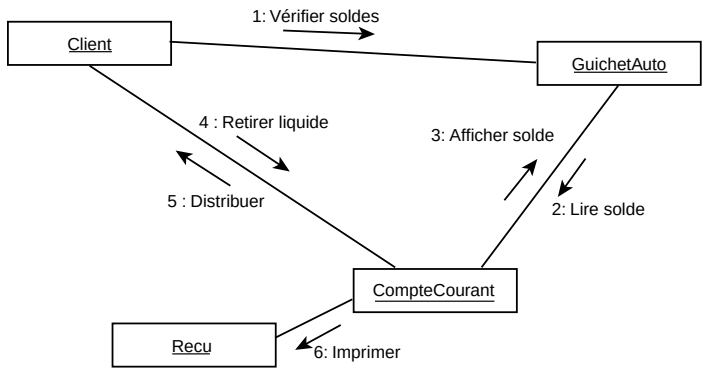


Ce diagramme montre l'interaction entre plusieurs classes. La classe `GuichetAuto` va déléguer à la classe `CompteCourant` toute la responsabilité de gestion du solde, alors que `CompteCourant` va déléguer à `GuichetAuto` la gestion de l'affichage à l'utilisateur.

Le diagramme de la Figure 11.21 est un *diagramme de séquence*. Il existe aussi des *diagrammes de collaboration* qui insistent sur les interactions intemporelles entre les classes

et que vous pouvez directement produire à partir du précédent, notamment avec Rational Rose (voir Figure 11.22).

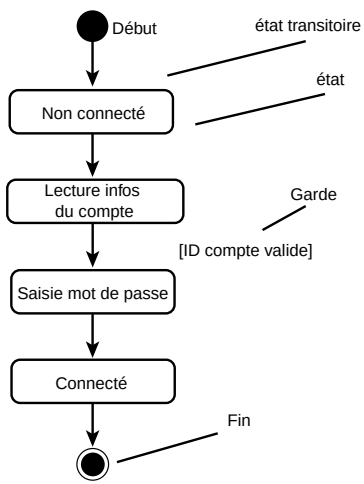
Figure 11.22
Diagramme de collaboration.



Diagrammes de transition d'état

Chaque objet a plusieurs *états* possibles, dont les transitions peuvent être modélisées dans un *diagramme d'état* ou *diagramme de transition* d'état. La Figure 11.23 montre les différents états de la classe `CompteClient` lors de la connexion du client au système.

Figure 11.23
États du compte client.

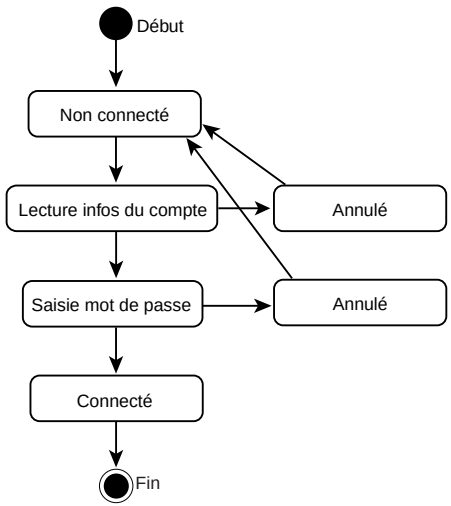


Chaque diagramme d'état commence par un simple état de `départ` et se termine avec zéro ou plusieurs états de `fin`. Les états individuels sont nommés et les transitions peuvent être étiquetées. Le `garde` indique une condition qui doit être satisfaite pour qu'un objet passe d'un état à un autre.

Super états

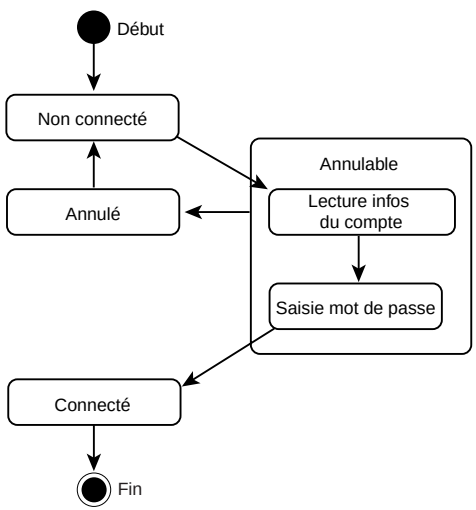
Le client peut changer d'avis à tout moment et décider de ne pas s'authentifier. Le système doit toujours accepter cette annulation et revenir à l'état "non authentifié" (voir Figure 11.24).

Figure 11.24
L'utilisateur peut annuler.



Le diagramme devient un peu plus compliqué, mais il peut être simplifié à l'aide d'un *super état* (voir Figure 11.25).

Figure 11.25
Super état.



Le diagramme de la Figure 11.25 fournit les mêmes informations que celui de la Figure 11.24, mais il est plus clair et plus facile à lire. Entre le moment de la demande d'authentification jusqu'à la fin de cette opération, le processus peut être annulé, vous revenez alors à l'état "non authentifié".

Étapes 4 à 6 – Implémentation, tests et déploiement

Les trois dernières étapes sont importantes, mais ne seront pas traitées ici. Si vous utilisez C++, l'implémentation sera détaillée dans le reste de cet ouvrage. Les tests et le déploiement sont deux disciplines complexes, qui n'entrent pas dans le cadre de ce livre. N'oubliez pas tout de même de tester soigneusement vos classes à la fois séparément et ensemble pour vérifier votre implémentation et votre conception.

Itérations

Dans le cadre de *Rational Unified Process*, les activités que nous venons d'énumérer sont appelées *workflow*s ("flux de travail") ; elles varient dans les phases de mise en route, d'élaboration, de construction et de transition. La modélisation d'activité est à son apogée lors de la mise en route, mais elle peut toujours survenir pendant la construction. L'implémentation, quant à elle, est forte pendant la construction, mais elle peut survenir au moment de la création des prototypes pour la phase d'élaboration.

Dans chaque phase, la construction par exemple, il peut y avoir plusieurs itérations. Dans la première itération de construction, il est ainsi possible de développer les fonctions clés du système. Dans la deuxième, on les approfondit et on en ajoute d'autres. L'approfondissement et l'ajout sont accentués dans la troisième, jusqu'à parvenir à une itération produisant le système complet.

Questions-réponses

Q Je n'ai appris aucun élément de programmation C++ dans ce chapitre. Quel est son objectif ?

R Pour rédiger des programmes C++ efficaces, vous devez savoir les structurer. En planifiant votre projet avant de commencer le codage, vous construirez de meilleurs programmes.

Q En quoi l'analyse et la conception orientées objet diffèrent-elles fondamentalement des autres approches ?

R Avec les autres techniques, les analystes et programmeurs tendaient à penser aux programmes comme à des groupes de fonctions agissant sur des données. Dans la

programmation orientée objet, les données et fonctionnalités sont des unités ayant à la fois la connaissance (les données) et les capacités (les fonctions). Les programmes procéduraux mettent l'accent sur les fonctions et sur la façon dont elles agissent sur les données. On dit souvent que les programmes écrits en Pascal ou en C sont des collections de procédures alors que ceux écrits en C++ sont des collections de classes.

Q La programmation orientée objet est-elle la solution miracle à tous les problèmes ?

R Non, mais en ce qui concerne les problèmes importants et complexes, l'analyse et la conception orientées objet fournissent des outils appréciables et inégalés.

Testez vos connaissances

1. Quelle est la différence entre la programmation orientée objet et la programmation procédurale ?
2. Quelles sont les phases de l'analyse et de la conception orientées objet ?
3. Qu'est-ce que l'encapsulation ?
4. Au regard de l'analyse, qu'est-ce qu'un domaine ?
5. Au regard de l'analyse, qu'est-ce qu'un acteur ?
6. Qu'est-ce qu'un cas d'utilisation ?
7. Laquelle des affirmations suivantes est vraie ?

   a.Un chat est une forme d'animal spécialisée.

   b.Un animal est une forme spécialisée de chat et de chien.

Exercices

1. Un système informatique est constitué de plusieurs éléments : clavier, souris, écran, unité centrale. Dessinez un diagramme de composition pour illustrer la relation entre l'ordinateur et ses éléments. Astuce : il s'agit d'une agrégation.
2. Supposons que vous deviez simuler l'intersection de deux rues avec des feux tricolores et des passages pour piétons. L'objectif est de déterminer si le chronométrage des feux permettra un flux normal de la circulation.

   Quelles sortes d'objets faut-il modéliser ? Quelles seraient les classes ?



12

Héritage

Au sommaire de ce chapitre

- Nature de l'héritage
- Comment utiliser l'héritage pour dériver d'une classe à partir d'une autre
- Rôle et utilisation des accès protégés
- Nature et rôle des fonctions virtuelles

Vous avez vu, au chapitre précédent, un certain nombre de concepts liés à la programmation orientée objet qui incluent les principes de spécialisation/généralisation implémentés en C++ par le biais de l'héritage.

Qu'est-ce que l'héritage ?

Lorsque vous regardez un chien, que voyez-vous ? Une tête sur quatre pattes. Un biologiste verra un ensemble d'organes, un physicien considérera qu'il s'agit d'atomes exerçant des forces, et un zoologiste le classera dans l'espèce *canine domesticus*.

Nous nous intéresserons ici à cette dernière façon de voir. Un chien fait partie de l'espèce canine, un canin est un mammifère, etc. Les zoologistes classent les êtres vivants en Règne, Lignée, Classe, Ordre, Famille, Genre et Espèce.

Cette hiérarchie de spécialisation/généralisation établit une relation *est-un*. Un Homo Sapiens (un homme) est une sorte de primate et cette relation apparaît partout : un monospace est une sorte de voiture, qui est elle-même une sorte de véhicule. Une glace est une sorte de dessert, qui est lui-même une sorte d'aliment.

Lorsque l'on peut dire que quelque chose est une sorte d'autre chose, cela signifie qu'il est une spécialisation de cette autre chose. Une voiture, par exemple, est un type spécial de véhicule.

Héritage et dérivation

Un chien hérite de – c'est-à-dire reçoit automatiquement – toutes les caractéristiques d'un mammifère. Étant un mammifère, vous savez donc qu'il se déplace et qu'il respire à l'aide de poumons car tous les mammifères, par définition, le font. Le concept de chien ajoute l'idée qu'il aboie, qu'il remue la queue, qu'il mange mes chaussons, etc.

Vous pouvez diviser les chiens en chien de travail, en chien de garde et en chien de chasse, et vous pouvez encore subdiviser les chiens de chasse en épagneuls, en retrievers, etc. Finalement, chacun d'entre eux peut encore être spécialisé ; les retrievers, par exemple, peuvent être divisés en labradors et en goldens.

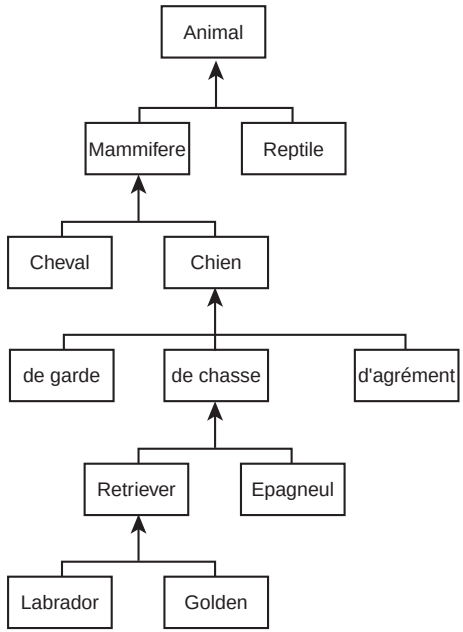
Un labrador est une sorte de retriever, qui est un chien de chasse, qui est un chien et donc une sorte de mammifère, qui est une sorte d'animal et, par conséquent, un être vivant. Cette hiérarchie est représentée par la Figure 12.1.

C++ essaie de représenter ce type de relations en définissant des classes qui dérivent les unes des autres. La dérivation est un moyen d'exprimer une relation *est-un*. Par exemple, la classe `Chien` dérive de la classe `Mammifere`. Ainsi, vous n'avez pas besoin de définir explicitement les mouvements des chiens, puisqu'ils héritent cette caractéristique des mammifères.

Une classe qui ajoute une fonctionnalité à une classe existante dérive de cette classe d'origine – qui est donc la classe de base de la nouvelle classe.

Si la classe `Chien` dérive de la classe `Mammifere`, `Mammifere` est la classe de base de `Chien`. Les classes dérivées sont des surensembles de leurs classes de base. De la même façon qu'un chien est un mammifère spécialisé, la classe `Chien` inclut des méthodes et des données supplémentaires par rapport à la classe `Mammifere`.

Figure 12.1
Hiérarchie d'animaux.



En général, une classe de base est associée à plusieurs classes dérivées. Les classes `Chien`, `Chat` et `Cheval` dérivent toutes de la classe `Mammifere`.

Le règne animal

Pour clarifier les concepts de dérivation et d'héritage, les différentes classes présentées dans ce chapitre portent des noms d'animaux. Supposez que vous deviez concevoir un jeu de simulation d'une exploitation agricole.

Vous allez développer un ensemble d'animaux domestiques, parmi lesquelles figurent des chevaux, des vaches, des cochons, des moutons, etc. Pour que les animaux réalisent des actions, vous implémentez des méthodes pour ces classes mais, pour le moment, vous allez vous créer de simples squelettes de fonctions qui se contenteront d'afficher un simple message.

Écrire des squelettes de fonctions signifie que vous écrirez le strict minimum pour montrer qu'elles sont appelées et que vous laisserez les détails pour plus tard, lorsque vous aurez plus de temps. Vous êtes évidemment libre d'étendre le code minimal de ce chapitre pour permettre aux animaux de se comporter plus naturellement.

Les exemples qui utilisent des animaux sont simples, mais ces concepts s'appliquent à d'autres domaines. Si vous construisez un distributeur automatique, par exemple, vous aurez un compte chèques, qui est un type de compte bancaire, qui est un type de compte.

Syntaxe de la dérivation

Pour déclarer une classe dérivée, vous devez indiquer le nom de la classe, suivi d'un caractère deux-points, du type de la dérivation (public, ou autre) et de la classe de base. Exemple :

```
class classeDerivee : typeAcces classeBase
```

Par exemple, pour créer une nouvelle classe appelée `Chien` qui hérite d'une classe `Mammifere` existante :

```
class Chien : public Mammifere
```

Les différents types de dérivations (`typeAcces`) sont décrits dans la suite du chapitre. Pour l'instant, nous utiliserons le plus courant : `public`. La classe de base doit avoir été déclarée avant la classe dérivée ; sinon, le compilateur sera incapable de créer le fichier exécutable. Dans le Listing 12.1, la classe `Chien` est dérivée de la classe `Mammifere`.

Listing 12.1 : Héritage simple

```
 1:  //Listing 12.1 Héritage simple
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  enum RACE { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
 6:
 7:  class Mammifere
 8:  {
 9:  public:
10:     // constructeurs
11:     Mammifere();
12:     ~Mammifere();
13:
14:     // méthodes d'accès
15:     int GetAge() const;
16:     void SetAge(int);
17:     int GetPoids() const;
18:     void SetPoids();
19:
20:     // autres méthodes
21:     void Crier() const;
22:     void Dormir() const;
```

```
23:
24:
25:  protected:
26:     int sonAge;
27:     int sonPoids;
28:  };
29:
30:  class Chien : public Mammifere
31:  {
32:  public:
33:
34:     // constructeurs
35:     Chien();
36:     ~Chien();
37:
38:     // méthodes d'accès
39:     RACE GetRace() const;
40:     void SetRace(RACE);
41:
42:     // autres méthodes
43:     RemuerQueue();
44:     Quemander();
45:
46:  protected:
47:     RACE saRace;
48:  };
```

Ce programme ne produit pas de résultat visible car il ne contient qu'un ensemble de déclarations de classes sans implémentation, mais cela n'enlève rien à son intérêt.

La classe `Mammifere` est déclarée de la ligne 6 à la ligne 27. Dans cet exemple, elle n'est pas dérivée d'une autre classe. Dans le monde réel, les mammifères dériveraient d'une autre classe puisqu'en réalité un mammifère est une sorte d'animal. En C++, vous ne pouvez représenter qu'une partie des informations relatives à un objet donné : la réalité est trop complexe pour être cernée dans tous ses détails et sa globalité. Une hiérarchie C++ est donc toujours une représentation partielle des informations disponibles. Pour créer efficacement des classes dérivées en C++, le meilleur moyen est de regrouper le plus de caractéristiques communes aux objets de la vie courante, sans ajouter de complexité inutile.

Il fallait une origine à la hiérarchie : nous l'avons trouvée dans la classe `Mammifere`. Pour cette raison, certaines variables membres qui devraient appartenir à des classes de base situées en amont sont représentées dans cette classe. Les animaux ont tous un poids et un âge, par exemple : si `Mammifere` dérivait de `Animal`, elle hériterait de ces attributs au lieu de les définir.

Si l'on devait plus tard ajouter un autre animal partageant certaines de ces fonctionnalités (par exemple `Insecte`), les attributs concernés pourraient être dirigés vers une nouvelle classe `Animal` qui deviendrait la classe de base de `Mammifere` et `Insecte`. C'est ainsi qu'évoluent les hiérarchies de classes.

Pour faciliter la compréhension et la gestion des applications, seules six méthodes ont été intégrées à la classe `Mammifere` : quatre méthodes publiques d'accès ainsi que `Crier()`, et `Dormir()`.

Comme l'indique la ligne 30, la classe `Chien` hérite de la classe `Mammifere`. Vous le savez grâce aux deux-points qui suivent le nom de la classe (`Chien`), suivis du nom de la classe de base (`Mammifere`).

Chaque objet `Chien` a trois variables membres : `sonAge`, `sonPoids` et `saRace`. Vous remarquerez que les variables `sonAge` et `sonPoids` ne figurent pas dans la déclaration de la classe `Chien` ; elles sont héritées de la classe `Mammifere`, tout comme l'opérateur de copie, les constructeurs et le destructeur.

Privé ou protégé ?

Vous avez certainement découvert un nouveau mot-clé aux lignes 24 et 45 du Listing 12.1 : il s'agit de `protected`. Jusqu'à présent, les données des classes étaient privées, mais les membres privés ne sont pas accessibles en dehors de la classe existante. Cette confidentialité empêche également l'accès à ces membres à partir des classes dérivées. Les variables `sonAge` et `sonPoids` pourraient être publiques, mais ce n'est pas souhaitable car vous ne voulez pas que d'autres classes accèdent directement à ces données.



Stroustrup (le créateur de C++), dans The Design and Evolution of C++, *énonce que toutes les données membres devraient être déclarées comme privées, jamais protégées. Les méthodes protégées, cependant, ne posent généralement pas de problèmes et peuvent être très utiles.*

En réalité, vous souhaitez les rendre visibles uniquement à cette classe et à ses classes dérivées et c'est exactement ce fait le mot-clé `protected`.

Il existe trois identificateurs d'accès : `public`, `protected` et `private`. Si un objet de votre classe est utilisé par une fonction, celui-ci permet d'accéder à toutes les données et fonctions membres publiques. À leur tour, les fonctions membres peuvent manipuler toutes les données et fonctions membres privées de leur classe et toutes les données et fonctions membres protégées de toutes les classes dont elles héritent. En conséquence, la fonction

`Chien::RemuerQueue()` a accès à la donnée privée `saRace` et aux données protégées `sonAge` et `sonPoids` de la classe `Mammifere`.

Même si d'autres classes s'intercalaient entre `Mammifere` et `Chien` (`AnimalDomestique`, par exemple), la classe `Chien` aurait quand même accès aux membres protégés de la classe `Mammifere` à condition que ces autres classes utilisent toutes un héritage public. Pour en savoir plus sur l'héritage privé, reportez-vous au Chapitre 16.

Le Listing 12.2 crée des objets de type `Chien`, puis accède aux données et aux méthodes membres de ces objets.

Listing 12.2 : Utilisation d'un objet dérivé

```
 1:  //Listing 12.2 Utilisation d'un objet dérivé
 2:  #include <iostream>
 3:  using std::cout;
 4:  using std::endl;
 5:
 6:  enum RACE { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
 7:
 8:  class Mammifere
 9:  {
10:  public:
11:     // constructeurs
12:     Mammifere():sonAge(2), sonPoids(5){}
13:     ~Mammifere(){}
14:
15:     // méthodes d'accès
16:     int GetAge() const   { return sonAge; }
17:     void SetAge(int age) { sonAge = age; }
18:     int GetPoids() const { return sonPoids; }
19:     void GetPoids(int Poids) { sonPoids = Poids; }
20:
21:     // autres méthodes
22:     void Crier()const
22a:        { cout <<"Le cri du mammifère !\n"; }
23:     void Dormir()const { cout << "Chut. Je dors.\n"; }
24:
25:  protected:
26:     int sonAge;
27:     int sonPoids;
28:  };
29:
30:  class Chien : public Mammifere
31:  {
32:  public:
33:
```

```
34:     // constructeurs
35:     Chien():saRace(GOLDEN){}
36:     ~Chien(){}
37:
38:     // méthodes d'accès
39:     RACE GetRace() const { return saRace; }
40:     void SetRace(RACE race) { saRace = race; }
41:
42:     // autres méthodes
43:     void RemuerQueue() const
43a:         { cout << "Je remue la queue...\n"; }
44:     void Quemander() const
44a:         { cout << "Je mendie de la nourriture...\n"; }
45:
46:  private:
47:     RACE saRace;
48:  };
49:
50:  int main()
51:  {
52:     Chien Fido;
53:     Fido.Crier();
54:     Fido.RemuerQueue();
55:     cout << "Fido a " << Fido.sonAge() << " ans" << endl;
56:     return 0;
57:  }
```

Ce programme produit le résultat suivant :

```
Le cri du mammifère !
Je remue la queue...
Fido a 2 ans
```

La classe `Mammifere` est déclarée de la ligne 8 à la ligne 28 (pour raccourcir le listing, les fonctions membres ont été définies `inline`). La classe `Chien` est une classe dérivée de la classe `Mammifere` (lignes 30 à 48), ce qui confère les mêmes attributs à tous les chiens : un âge, un poids et une race. Comme on l'a indiqué plus haut, les deux premiers proviennent de la classe de base, `Mammifere`.

La ligne 52 déclare un objet `Chien` nommé `Fido`. Il hérite de tous les attributs de la classe `Mammifere` et de la classe `Chien`. Ainsi, `Fido` peut remuer la queue, crier et dormir, puisqu'il dispose des fonctions `RemuerQueue()`, `Crier()` et `Dormir()`. Aux lignes 53 et 54, `Fido` appelle deux de ces méthodes de la classe de base `Mammifere`. La ligne 55 appelle la méthode d'accès `LireAge()` de la classe de base.

L'héritage avec les constructeurs et les destructeurs

Par héritage, les objets `Chien` sont également des objets `Mammifere`. Lorsque `Fido` est créé, son constructeur de base est donc appelé pour produire un objet `Mammifere`. Puis, le constructeur de `Chien` est appelé à son tour pour compléter l'objet `Fido`. Comme on n'a pas fourni de paramètre au constructeur de `Chien`, c'est le constructeur par défaut qui est appelé dans les deux cas. `Fido` ne peut exister tant qu'il n'a pas été totalement construit, ce qui signifie que sa partie et sa partie `Chien` doivent avoir été construites. C'est pour cette raison que les deux constructeurs doivent être appelés.

Lorsque l'objet `Fido` est supprimé, le destructeur de `Chien` est d'abord appelé, puis c'est le tour du destructeur de la partie `Mammifere`. Chacun d'eux permet de nettoyer les attributs de sa propre partie de `Fido`. Le Listing 12.3 met en évidence les appels aux constructeurs et aux destructeurs.

Listing 12.3 : Appel des constructeurs et des destructeurs

```
 1:  //Listing 12.3 Appel des constructeurs et des destructeurs.
 2:  #include <iostream>
 3:  using namespace std;
 4:  enum RACE { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
 5:
 6:  class Mammifere
 7:  {
 8:  public:
 9:      // constructeurs
10:      Mammifere();
11:     ~Mammifere();
12:
13:     // méthodes d'accès
14:     int GetAge() const { return sonAge; }
15:     void SetAge(int age) { sonAge = age; }
16:     int GetPoids() const { return sonPoids; }
17:     void SetPoids(int Poids) { sonPoids = Poids; }
18:
19:     // autres méthodes
20:     void Crier() const { cout << "Le cri du mammifère !\n"; }
21:     void Dormir() const { cout << "Chut. Je dors.\n"; }
22:
23:  protected:
24:     int sonAge;
25:     int sonPoids;
26:  };
27:
28:  class Chien : public Mammifere
```

```
29:  {
30:  public:
31:
32:     // constructeurs
33:     Chien();
34:     ~Chien();
35:
36:     // méthodes d'accès
37:     RACE GetRace() const { return saRace; }
38:     void SetRace(RACE race) { saRace = race; }
39:
40:     // autres méthodes
41:     void RemuerQueue() const
41a           { cout << "Je remue la queue...\n"; }
42:     void Quemander() const
42a:          { cout << "Je mendie de la nourriture...\n"; }
43:
44:  private:
45:     RACE saRace;
46:  };
47:
48:  Mammifere::Mammifere():
49:    sonAge(3),
50:    sonPoids(5)
51:  {
52:     std::cout << "Constructeur de Mammifere... " << endl;
53:  }
54:
55:  Mammifere::~Mammifere()
56:  {
57:     std::cout << "Destructeur de Mammifere... " << endl;
58:  }
59:
60:  Chien::Chien():
61:    saRace(GOLDEN)
62:  {
63:     std::cout << "Constructeur de Chien... " << endl;
64:  }
65:
66:  Chien::~Chien()
67:  {
68:     std::cout << "Destructeur de Chien... " <<endl;
69:  }
70:  int main()
71:  {
72:     Chien Fido;
73:     Fido.Crier();
```

```
74:     Fido.RemuerQueue();
75:     std::cout << "Fido a " << Fido.GetAge() << " ans" << endl;
76:     return 0;
77:  }
```

Ce programme produit le résultat suivant :

```
Constructeur de Mammifere...
Constructeur de Chien...
Le cri du mammifère !
Je remue la queue...
Fido a 3 ans
Destructeur de Chien...
Destructeur de Mammifere...
```

Ce listing ressemble beaucoup au Listing 12.2, mais les lignes 48 à 69 signalent désormais les appels aux constructeurs et destructeurs à l'écran. Le constructeur de `Mammifere` est appelé avant le constructeur de `Chien`. La classe `Chien` est alors complètement définie et les méthodes sont implémentées. Lorsque `Fido` devient hors de portée, le destructeur de `Chien`, puis celui de `Mammifere` sont appelés. Cela est confirmé par le résultat du listing.

Passer des paramètres aux constructeurs de base

Vous souhaiterez peut-être initialiser certaines valeurs dans un constructeur de base. Vous pourriez, par exemple, surcharger le constructeur de `Mammifere` et le constructeur de `Chien` pour utiliser respectivement un âge et une race spécifiques. Comment faire pour passer l'âge et le poids au bon constructeur de `Mammifere` ? Comment faire pour initialiser le poids d'un objet `Chien` si cette opération n'est pas prévue dans la classe `Mammifere` ?

L'initialisation de la classe de base peut avoir lieu pendant celle de la classe dérivée, en précisant le nom de la classe de base, suivi des paramètres voulus. Le Listing 12.4 montre comment faire.

Listing 12.4 : Surcharge de constructeurs dans des classes dérivées

```
1:     // Listing 12.4 Surcharge des constructeurs
1a:    // dans des classes dérivées
2:  #include <iostream>
3:  using namespace std;
4:
5:  enum RACE { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7:  class Mammifere
8:  {
9:  public:
```

```
10:     // constructeurs
11:     Mammifere();
12:     Mammifere(int age);
13:     ~Mammifere();
14:
15:     // méthodes d'accès
16:     int GetAge() const { return sonAge; }
17:     void SetAge(int age) { sonAge = age; }
18:     int GetPoids() const { return sonPoids; }
19:     void SetPoids(int Poids) { sonPoids = Poids; }
20:
21:     //autres méthodes
22:     void Crier() const
22a:          { cout << "Le cri du mammifère !\n"; }
23:     void Dormir() const { cout << "Chut. Je dors.\n"; }
24:
25:
26:  protected:
27:     int sonAge;
28:     int sonPoids;
29:  };
30:
31:  class Chien : public Mammifere
32:  {
33:  public:
34:
35:     // constructeurs
36:     Chien();
37:     Chien(int age);
38:     Chien(int age, int poids);
39:     Chien(int age, RACE race);
40:     Chien(int age, int poids, RACE race);
41:     ~Chien();
42:
43:     //  méthodes d'accès
44:     RACE GetRace() const { return saRace; }
45:     void SetRace(RACE race) { saRace = race; }
46:
47:     // autres méthodes
48:     void RemuerQueue() const
48a:          { cout << "Je remue la queue...\n"; }
49:     void Quemander() const
49a:          { cout << "Je mendie de la nourriture...\n"; }
50:
51:  private:
52:     RACE saRace;
53:  };
```

```
54:
55:  Mammifere::Mammifere():
56:    sonAge(1),
57:    sonPoids(5)
58:  {
59:     cout << "Constructeur de Mammifere..." << endl;
60:  }
61:
62:  Mammifere::Mammifere(int age):
63:    sonAge(age),
64:    sonPoids(5)
65:  {
66:     cout << "Constructeur de Mammifere(int)..." << endl;
67:  }
68:
69:  Mammifere::~Mammifere()
70:  {
71:     cout << "Destructeur de Mammifere..." << endl;
72:  }
73:
74:  Chien::Chien():
75:    Mammifere(),
76:    saRace(GOLDEN)
77:  {
78:     cout << "Constructeur de Chien..." << endl;
79:  }
80:
81:  Chien::Chien(int age):
82:    Mammifere(age),
83:    saRace(GOLDEN)
84:  {
85:     cout << "Constructeur de Chien(int)..." << endl;
86:  }
87:
88:  Chien::Chien(int age, int poids):
89:    Mammifere(age),
90:    saRace(GOLDEN)
91:  {
92:     sonPoids = poids;
93:     cout << "Constructeur de Chien(int, int)..." << endl;
94:  }
95:
96:  Chien::Chien(int age, int poids, RACE race):
97:    Mammifere(age),
98:    saRace(race)
99:  {
```

```
100:     sonPoids = Poids;
101:     cout << "Constructeur de Chien(int, int, RACE)..." << endl;
102:  }
103:
104:  Chien::Chien(int age, RACE race):
105:    Mammifere(age),
106:    saRace(race)
107:  {
108:     cout << "Constructeur de Chien(int, RACE)..." << endl;
109:  }
110:
111:  Chien::~Chien()
112:  {
113:     cout << "Destructeur de Chien..." << endl;
114:  }
115:  int main()
116:  {
117:     Chien Fido;
118:     Chien Rover(5);
119:     Chien Buster(6, 8);
120:     Chien Yorkie (3, GOLDEN);
121:     Chien Dobbie (4, 20, DOBERMAN);
122:     Fido.Crier();
123:     Rover.RemuerQueue();
124:     cout << "Yorkie a " << Yorkie.GetAge()
125:          << " ans" << endl;
126:     cout << "Dobbie pèse ";
127:     cout << Dobbie.GetPoids() << " kilos" << endl;
128:     return 0;
129:  }
```



Le résultat obtenu a été numéroté pour faciliter l'analyse du code.

Ce programme produit le résultat suivant :

```
1:  Constructeur de Mammifere...
2:  Constructeur de Chien...
3:  Constructeur de Mammifere(int)...
4:  Constructeur de Chien(int)...
5:  Constructeur de Mammifere(int)...
6:  Constructeur de Chien(int, int)...
7:  Constructeur de Mammifere(int)...
8:  Constructeur de Chien(int, RACE)....
9:  Constructeur de Mammifere(int)...
```

```
10: Constructeur de Chien(int, int, RACE)...
11: Le cri du mammifère !
12: Je remue la queue...
13: Yorkie a 3 ans.
14: Dobbie pèse 20 kilos.
15: Destructeur de Chien...
16: Destructeur de Mammifere...
17: Destructeur de Chien...
18: Destructeur de Mammifere...
19: Destructeur de Chien...
20: Destructeur de Mammifere...
21: Destructeur de Chien...
22: Destructeur de Mammifere...
23: Destructeur de Chien...
24: Destructeur de Mammifere...
```

Le constructeur de `Mammifere` a été surchargé à la ligne 12 pour prendre en charge un entier (l'âge de l'animal). La variable `sonAge` est initialisée avec la valeur passée au constructeur et `sonPoids` est définie à 5 (lignes 62 à 67).

`Chien` surcharge cinq constructeurs (lignes 36 à 40). Le premier est le constructeur par défaut. À la ligne 37, le deuxième prend un âge, qui est le même paramètre que celui du constructeur de `Mammifere`. Le troisième constructeur prend à la fois un âge et un poids, le quatrième un âge et une race ; enfin, le dernier prend un âge, un poids et une race.

La ligne 74 contient le code du constructeur par défaut de `Chien` et vous pouvez constater que quelque chose a changé. Lorsque ce constructeur est appelé, il invoque à son tour le constructeur par défaut de `Mammifere`, comme on le voit à la ligne 75. Bien que ce ne soit pas strictement obligatoire de procéder ainsi, cela permet d'indiquer explicitement que l'on appelle le constructeur sans paramètre de la classe de base. Ce constructeur serait automatiquement appelé de toutes façons, mais cette façon de procéder explicite clairement vos intentions.

L'implémentation du constructeur de `Chien`, qui prend une valeur entière, figure de la ligne 81 à la ligne 86. Lors de l'étape d'initialisation (lignes 82 et 83), `Chien` initialise sa classe de base en lui passant le paramètre, puis initialise sa race (`saRace`).

Le deuxième constructeur de `Chien` est implémenté de la ligne 88 à la ligne 94. Il prend en charge deux paramètres. Cette fois encore, la classe de base est initialisée à l'aide du constructeur approprié (ligne 89) mais on en profite également pour affecter un poids à la variable `sonPoids` de la classe de base. Vous ne pouvez pas effectuer cette opération lors de la phase d'initialisation car `Mammifere` ne possède pas de constructeur prenant un poids en paramètre : c'est la raison pour laquelle on initialise cette variable dans le corps du constructeur de `Chien`.

Examinez le fonctionnement des autres constructeurs pour être certain de bien le comprendre. Distinguez notamment ce qui est initialisé lors de la phase d'initialisation de ce qui doit être traité dans le corps du constructeur.

Le résultat obtenu a été numéroté afin de pouvoir faire référence à chacune des lignes. Les deux premières correspondent à la création de l'instance de `Fido`, à l'aide du constructeur par défaut.

Les lignes 3 et 4 de la sortie correspondent à la création de `Rover`, alors que `Buster` est représenté aux lignes 5 et 6. Le constructeur de `Mammifere` appelé prend en paramètre un seul entier, alors que le constructeur de `Chien` prend deux valeurs entières.

Une fois que les objets ont été créés, ils sont traités puis détruits. Pour cela, le destructeur de `Chien` est appelé avant celui de `Mammifere`, cinq fois chacun.

Redéfinition des méthodes de la classe de base

Un objet `Chien` a accès à toutes les données et fonctions membres de la classe `Mammifere`, ainsi qu'à ses propres fonctions et données membres, comme `RemuerQueue()`. Une classe dérivée peut également redéfinir une méthode de la classe de base. La *rédéfinition d'une méthode* modifie l'implémentation d'une méthode de la classe de base dans une classe dérivée.

Quand on redéfinit une méthode, sa signature doit correspondre à celle de la méthode de la classe de base. Une signature de méthode est son prototype, sans le type du résultat – elle est donc formée du nom de la méthode, de la liste de ses paramètres et du mot `const` s'il est présent. Le résultat d'une méthode redéfinie peut donc être différent de celui de la méthode dans la classe de base.

Dans le Listing 12.5, la classe `Chien` redéfinit la méthode `Crier()` de la classe `Mammifere`. Pour des raisons d'espace, les fonctions d'accès ont été volontairement omises.

Listing 12.5 : Redéfinition d'une méthode de classe de base dans une classe dérivée

```
1:  // Listing 12.5 Redéfinition d'une méthode de
1a: // classe de base dans une classe dérivée
2:  #include <iostream>
3:  using std::cout;
4:
5:  enum RACE { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
6:
7:  class Mammifere
8:  {
9:  public:
```

```
10:     // constructeur et destructeur
11:     Mammifere() { cout << "Constructeur de Mammifere...\n"; }
12:     ~Mammifere() { cout << "Destructeur de Mammifere...\n"; }
13:
14:     // autres méthodes
15:     void Crier()const
15a:          { cout << "Le cri du mammifère !\n"; }
16:     void Dormir()const { cout << "Chut. Je dors.\n"; }
17:
18:  protected:
19:     int sonAge;
20:     int sonPoids;
21:  };
22:
23:  class Chien : public Mammifere
24:  {
25:  public:
26:     // constructeur et destructeur
27:     Chien(){ cout << "Constructeur de Chien...\n"; }
28:     ~Chien(){ cout << "Destructeur de Chien...\n"; }
29:
30:     // autres méthodes
31:     void RemuerQueue() const
31a:          { cout << "Je remue la queue...\n"; }
32:     void Quemander() const
32a:          { cout << "Je mendie de la nourriture...\n"; }
33:     void Crier() const { cout << "Ouah Ouah !\n"; }
34:
35:  private:
36:     RACE saRace;
37:  };
38:
39:  int main()
40:  {
41:     Mammifere grosAnimal;
42:     Chien Fido;
43:     grosAnimal.Crier();
44:     Fido.Crier();
45:     return 0;
46:  }
```

Ce programme produit le résultat suivant :

```
Constructeur de Mammifere...
Constructeur de Mammifere...
Constructeur de Chien...
Le cri du mammifère !
```

```
Ouah Ouah !
Destructeur de Chien...
Destructeur de Mammifere...
Destructeur de Mammifere...
```

On voit une méthode `Crier()` définie à la ligne 15 dans la classe `Mammifere`. La classe `Chien` déclarée aux lignes 23 à 37 hérite de `Mammifere` (ligne 23) et a donc accès à cette méthode `Crier()`. Elle la redéfinit toutefois ligne 33 afin que les objets `Chien` aboient lors de l'appel à cette méthode. La ligne 41 crée l'objet `Mammifere grosAnimal`, comme le montre la première ligne du résultat. La ligne suivante crée `Fido`, ce qui déclenche l'appel du constructeur de `Mammifere` et du constructeur de `Chien`.

À la ligne 43, l'objet `grosAnimal` appelle sa méthode `Crier()`, puis c'est au tour de l'objet `Chien` d'appeler sa méthode `Crier()`. L'affichage du programme reflète bien l'appel aux deux méthodes. Enfin, les deux objets sont supprimés à l'aide de leurs destructeurs respectifs.



Vous ne devez pas confondre les termes surcharge et redéfinition. Lorsque vous surchargez une méthode, vous créez plusieurs méthodes de même nom avec des signatures différentes dans la même classe. La redéfinition d'une méthode consiste à créer dans une classe dérivée une méthode de même nom et de même signature que celle de la classe de base.

Masquage de la méthode de la classe de base

Dans le listing précédent, la méthode `Crier()` de la classe `Chien` cachait la méthode de la classe de base. C'est ce que nous voulions, mais cette opération risque de produire des résultats inattendus. S'il existe une méthode `Bouger()` surchargée dans la classe `Mammifere` et que la classe `Chien` la redéfinit, la méthode de la classe `Chien` masquera *toutes* les méthodes portant le même nom dans la classe `Mammifere`.

Supposons que la classe `Mammifere` surcharge la méthode `Bouger()` à l'aide de trois fonctions – la première n'attend aucun paramètre, la deuxième attend un entier et la dernière attend un entier et une direction – et que la classe `Chien` redéfinit simplement la méthode `Bouger()` qui ne prend pas de paramètre. Il sera alors difficile d'avoir accès aux deux autres méthodes depuis un objet `Chien`. Le Listing 12.6 illustre ce problème.

Listing 12.6 : Masquage de méthodes

```
1:  //Listing 12.6 Masquage de méthodes
2:  #include <iostream>
3:  using std::cout;
4:
```

```
 5:  class Mammifere
 6:  {
 7:  public:
 8:       void Bouger() const
8a:        { cout << "Les mammifères se déplacent d'un pas\n"; }
 9:      void Bouger(int distance) const
10:      {
11:         cout << "Les mammifères se deplacent de ";
12:         cout << distance <<" pas.\n";
13:      }
14:  protected:
15:      int sonAge;
16:      int sonPoids;
17:  };
18:
19:  class Chien : public Mammifere
20:  {
21:  public:
22:      // Avertissement possible signalant
22a:     // que vous masquez une fonction!
23:      void Bouger() const
23a:         { cout << "Le chien se déplace de 5 pas.\n"; }
24:  };
25:
26:  int main()
27:  {
28:     Mammifere grosAnimal;
29:     Chien Fido;
30:     grosAnimal.Bouger();
31:     grosAnimal.Bouger(2);
32:     Fido.Bouger();
33:     // Fido.Bouger(10);
34:     return 0;
35:  }
```

Ce programme produit le résultat suivant :

```
Les mammifères se déplacent d'un pas
Les mammifères se déplacent de 2 pas
Le chien se déplace de 5 pas
```

Toutes les méthodes et les données non indispensables ont été supprimées de ces classes. Le programme commence par la déclaration des méthodes `Bouger()` surchargées dans la classe `Mammifere` (lignes 8 et 9). À la ligne 23, la classe `Chien` redéfinit la version de la fonction `Bouger()` sans paramètre. Les appels à ces méthodes (lignes 30 à 32) sont signalés dans le résultat.

Vous pouvez constater que la ligne 33 a été mise en commentaire (désactivée) car elle produit une erreur de compilation. Après avoir redéfini l'une des méthodes, vous ne pouvez plus utiliser aucune des méthodes de base du même nom. Bien qu'un objet `Chien` aurait pu appeler la méthode `Bouger(int)` s'il n'avait pas redéfini la méthode `Bouger()` sans paramètre, il doit désormais redéfinir les deux s'il compte utiliser les deux. Sinon, il *cache* la méthode qui n'est pas redéfinie. Ce comportement est analogue à la règle qui veut que si vous fournissez un constructeur quelconque, le compilateur ne vous fournira plus de constructeur par défaut.

La règle est donc la suivante : si vous redéfinissez une méthode surchargée, tous les autres surcharges de cette méthode sont masquées. Si tel n'est pas votre souhait, vous devez toutes les redéfinir.

Les programmeurs débutants oublient souvent le mot-clé `const` lorsqu'ils redéfinissent une méthode de la classe de base et la masque donc involontairement. En effet, le mot-clé `const` fait partie de la signature : l'oublier revient à modifier la signature et donc à cacher la méthode, au lieu de la redéfinir.

Redéfinir ou cacher

La section qui suit décrit les méthodes virtuelles. La redéfinition d'une méthode virtuelle autorise le polymorphisme, alors que le masquage d'une méthode le détériore. Vous allez très vite en apprendre davantage.

Appel de la méthode de base

Si vous avez redéfini une méthode de la classe de base, il est toujours possible de l'appeler en indiquant son nom complet, après le nom de la classe de base :

```
classeBase::Methode()
```

Vous pouvez donc appeler la méthode `Bouger()` de la classe `Mammifere` comme suit :

```
Mammifere::Bouger().
```

Vous pouvez utiliser ces noms qualifiés comme avec tout autre nom de méthode. Dans le Listing 12.6, il aurait donc été possible de modifier la ligne 33, de sorte que le programme puisse être compilé :

```
Fido.Mammifere::Bouger(10);
```

Cette ligne appelle explicitement la méthode de la classe `Mammifere`. Le Listing 12.7 exploite cette syntaxe.

Listing 12.7 : Appel d'une méthode de base depuis une méthode redéfinie

```
 1:  // Listing 12.7 Appel d'une méthode de base
 1a: // depuis une méthode redéfinie.
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Mammifere
 6:  {
 7:  public:
 8:      void Bouger() const
8a:       { cout << "Les mammifères se déplacent d'un pas\n"; }
 9:     void Bouger(int distance) const
10:     {
11:        cout << "Les mammifères se déplacent de " << distance;
12:        cout << " pas." << endl;
13:     }
14:
15:  protected:
16:     int sonAge;
17:     int sonPoids;
18:  };
19:
20:  class Chien : public Mammifere
21:  {
22:  public:
23:     void Bouger() const;
24:  };
25:
26:  void Chien::Bouger() const
27:  {
28:     cout << "Méthode Bouger de Chien...\n";
29:     Mammifere::Bouger(3);
30:  }
31:
32:  int main()
33:  {
34:     Mammifere grosAnimal;
35:     Chien Fido;
36:     grosAnimal.Bouger(2);
37:     Fido.Mammifere::Bouger(6);
38:     return 0;
39:  }
```

Ce programme produit le résultat suivant :

```
Les mammifères se déplacent de 2 pas
Les mammifères se déplacent de 6 pas
```

Les lignes 34 et 35 créent les objets `grosAnimal` et `Fido`, respectivement de la classe `Mammifere` et de la classe `Chien`. La méthode appelée à la ligne 36 est la méthode `Bouger()` de la classe `Mammifere` qui attend un paramètre entier.

Dans le listing précédent, le programmeur souhaitait également faire bouger l'objet `Chien` en appelant la fonction `Bouger(int)` de la classe `Chien`, or cette classe a redéfini la méthode (sans paramètre) sans redéfinir également celle qui en attend un – elle ne dispose donc pas d'une version de cette méthode prenant un entier en paramètre. Pour résoudre ce problème, la ligne 37 appelle explicitement la méthode `Bouger(int)` de la classe de base.



Lors de l'appel des méthodes redéfinies des classes ancêtres à l'aide de "::", si une nouvelle classe est insérée dans la hiérarchie d'héritage entre le descendant et son ancêtre, le descendant fera un donc appel qui ignorera la classe intermédiaire et risque de ne pas appeler certaines capacités essentielles, implémentées par celle-ci.

Faire	*Ne pas faire*
● Dériver des classes existantes et bien testées pour en accroître les fonctionnalités. ● Redéfinir les méthodes de la classe de base pour modifier le comportement de certaines méthodes dans la classe dérivée.	● Masquer une fonction de la classe de base en modifiant sa signature. ● Oublier que `const` fait partie de la signature. ● Oublier que le type du résultat ne fait pas partie de la signature.

Méthodes virtuelles

Nous avons vu qu'un objet `Chien` *est-un* objet `Mammifere`. Jusqu'à maintenant, cela signifiait simplement qu'il héritait de tous les attributs (données) et de toutes les fonctionnalités (méthodes) de sa classe de base. Cependant, en C++ la relation *est-un* va beaucoup plus loin que cela.

En C++, il est en effet possible d'étendre le polymorphisme pour permettre aux pointeurs vers les classes de base de recevoir des objets des classes dérivées. On peut donc écrire :

```
Mammifere* pMammifere = new Chien;
```

Cette instruction crée un nouvel objet `Chien` sur le tas et renvoie un pointeur vers cet objet, que l'on peut affecter à un pointeur sur un objet `Mammifere`. Ceci est possible car un chien est un mammifère.



Il s'agit de l'essence même du polymorphisme. Par exemple, vous pouvez créer plusieurs types de fenêtres (boîtes de dialogue, fenêtres de défilement, zones de listes, etc.) et associer à chacune d'elles une fonction virtuelle `dessiner()`*. En affectant les types dérivés à un pointeur de fenêtre, le programme est capable d'appeler la fonction* `dessiner()` *appropriée, quel que soit le type de fenêtre traité.*

Vous pouvez ensuite utiliser ce pointeur pour appeler n'importe quelle méthode de la classe `Mammifere`. L'objectif est que les méthodes redéfinies dans la classe `Chien()` appellent la fonction adéquate et c'est justement ce que permettent les fonctions virtuelles. Pour créer une fonction virtuelle, ajoutez le mot-clé `virtual` devant sa déclaration.

Le Listing 12.8 illustre parfaitement ce processus.

Listing 12.8 : Utilisation de fonctions virtuelles

```
 1:  //Listing 12.8 Utilisation de fonctions virtuelles
 2:  #include <iostream>
 3:  using std::cout;
 4:
 5:  class Mammifere
 6:  {
 7:  public:
 8:      Mammifere():wAge(1)
8a:        { cout << "Constructeur de Mammifere...\n"; }
 9:      virtual ~Mammifere()
9a:        { cout << "Destructeur de Mammifere...\n"; }
10:     void Bouger() const
10a:       { cout << "Les mammifères se déplacent d'un pas\n"; }
11:     virtual void Crier() const
11a:       { cout << "Le cri du mammifère !\n"; }
12:
13:  protected:
14:     int sonAge;
15:  };
16:
17:  class Chien : public Mammifere
18:  {
19:  public:
20:     Chien() { cout << "Constructeur de Chien...\n"; }
21:     virtual ~Chien() { cout << "Destructeur de Chien...\n"; }
22:     void RemuerQueue() { cout << "Je remue la queue...\n"; }
23:     void Crier()const { cout << "Ouah Ouah !\n"; }
24:     void Bouger()const
24a:       { cout << "Le chien se déplace de 5 pas...\n"; }
25:  };
```

```
26:
27:  int main()
28:  {
29:     Mammifere *pChien = new Chien;
30:     pChien->Bouger();
31:     pChien->Crier();
32:
33:     return 0;
34:  }
```

Ce programme produit le résultat suivant :

```
Constructeur de Mammifere...
Constructeur de Chien...
Les mammifères se deplacent d'un pas
Ouah Ouah !
```

La ligne 11 définit la méthode virtuelle `Crier()` dans la classe `Mammifere`. Le concepteur de cette classe indique donc qu'il s'attend à ce qu'elle devienne la classe de base d'une autre et que la classe dérivée voudra sûrement redéfinir cette méthode

La ligne 29 crée un pointeur vers un `Mammifere (pChien)`, mais lui affecte l'adresse d'un nouvel objet `Chien`. Cette affectation est autorisée puisqu'un chien *est-un* mammifère. La ligne 30 se sert ensuite de ce pointeur pour appeler la méthode `Bouger()`. Comme il sait que `pChien` est un `Mammifere`, le compilateur examine l'objet `Mammifere` pour trouver sa méthode `Bouger()`. La ligne 10 montre qu'il s'agit d'une méthode classique, non virtuelle, et c'est la raison pour laquelle il appelle la version de `Mammifere`.

La ligne 31 appelle la méthode `Crier()` sur ce même pointeur. Celle-ci étant virtuelle (ligne 11), c'est la méthode redéfinie dans la classe `Chien` qui est cette fois-ci invoquée.

Tout ceci a un aspect presque magique. La fonction appelante sait qu'elle dispose d'un pointeur vers un `Mammifere`, pourtant c'est une méthode de `Chien` qui est appelée. En fait, si l'on avait un tableau de pointeurs de `Mammifere`, pointant chacun sur une sous-classe différente de celle-ci, on pourrait les appeler à tour de rôle et, à chaque fois, ce serait la méthode appropriée qui serait invoquée. C'est ce que l'on fait dans le Listing 12.9.

Listing 12.9 : Appels consécutifs de plusieurs fonctions virtuelles

```
1:  //Listing 12.9 Appels consécutifs de
1a: // plusieurs fonctions virtuelles
2:  #include <iostream>
3:  using namespace std;
4:
5:  class Mammifere
6:  {
```

```
 7:  public:
 8:     Mammifere():sonAge(1) {  }
 9:     virtual ~Mammifere() { }
10:     virtual void Crier() const
10a:          { cout << "Le cri du mammifère !\n"; }
11:
12: protected:
13:    int sonAge;
14:  };
15:
16:  class Chien : public Mammifere
17:  {
18:  public:
19:     void Crier() const { cout << "Ouah Ouah!\n"; }
20:  };
21:
22:  class Chat : public Mammifere
23:  {
24:  public:
25:     void Crier() const { cout << "Miaou !\n"; }
26:  };
27:
28:
29:  class Cheval : public Mammifere
30:  {
31:  public:
32:     void Crier() const { cout << "Hihiiiii !\n"; }
33:  };
34:
35:  class Cochon : public Mammifere
36:  {
37:  public:
38:     void Crier() const { cout << "Grouiiiik !\n"; }
39:  };
40:
41:  int main()
42:  {
43:     Mammifere* Tableau[5];
44:     Mammifere* ptr;
45:     int choix, i;
46:     for ( i = 0; i < 5; i++)
47:     {
48:        cout <<
48a:       "(1)Chien (2)Chat (3)Cheval (4)Cochon - Votre choix : ";
49:        cin >> choix;
50:        switch (choix)
51:        {
```

```
52:         case 1: ptr = new Chien;
53:               break;
54:         case 2: ptr = new Chat;
55:               break;
56:         case 3: ptr = new Cheval;
57:               break;
58:         case 4: ptr = new Cochon;
59:               break;
60:         default: ptr = new Mammifere;
61:               break;
62:         }
63:         Tableau[i] = ptr;
64:      }
65:      for (i = 0; i < 5; i++)
66:         Tableau[i]->Crier();
67:      return 0;
68:   }
```

Ce programme produit le résultat suivant :

```
(1)Chien (2)Chat (3)Cheval (4)Cochon - Votre choix : 1
(1)Chien (2)Chat (3)Cheval (4)Cochon - Votre choix : 2
(1)Chien (2)Chat (3)Cheval (4)Cochon - Votre choix : 3
(1)Chien (2)Chat (3)Cheval (4)Cochon - Votre choix : 4
(1)Chien (2)Chat (3)Cheval (4)Cochon - Votre choix : 5
Ouah Ouah !
Miaou !
Hihiiiii !
Grouiiiik !
Le cri du mammifère !
```

Ce programme allégé illustre les méthodes virtuelles sous leur forme la plus pure. Les quatre classes qui sont déclarées (`Chien`, `Chat`, `Cheval` et `Cochon`) dérivent toutes de `Mammifere`.

La ligne 10 déclare la fonction membre `Crier()` de la classe `Mammifere` comme virtuelle. Les quatre classes dérivées redéfinissent toutes cette méthode (lignes 19, 25, 32 et 38).

Le programme boucle cinq fois entre les lignes 46 et 64. À chaque itération, l'utilisateur est invité à choisir un objet à créer et l'on ajoute automatiquement au tableau un nouveau pointeur vers ce type d'objet grâce à l'instruction `switch` des lignes 50 à 62.

Au moment de la compilation, il est donc impossible de connaître les objets qui seront créés et par conséquent les méthodes `Crier()` qui seront appelées. Le pointeur `ptr` est associé à son objet au moment de l'exécution : c'est ce que l'on appelle une *liaison dynamique* (ou *liaison retardée*), par opposition à la liaison *statique* (qui à lieu au moment de la compilation).

Le programme boucle à nouveau sur le tableau entre les lignes 65 et 66. Cette fois-ci, chaque objet appelle sa méthode `Crier()`. Celle-ci étant virtuelle dans la classe de base, ce seront les méthodes `Crier()` appropriées à chaque type qui seront appelées. Vous voyez dans le résultat que si vous choisissez des types différents, la méthode correspondante est appelée à chaque fois.

Le fait de marquer une méthode membre comme virtuelle dans la classe de base implique-t-il de devoir faire de même dans les classes dérivées ?

Non, lorsqu'une méthode est virtuelle, elle le reste si vous la redéfinissez dans les classes dérivées. Il est cependant conseillé (bien que non obligatoire) de la signaler comme virtuelle, afin de rendre le code plus compréhensible.

Fonctionnement des méthodes virtuelles

Lorsqu'on crée un objet dérivé comme un objet `Chien`, on appelle d'abord le constructeur de la classe de base puis le constructeur de la classe dérivée. La Figure 12.2 présente l'objet `Chien` après sa création : vous remarquerez que la partie `Mammifere` de l'objet se situe en mémoire à côté de la partie `Chien`.

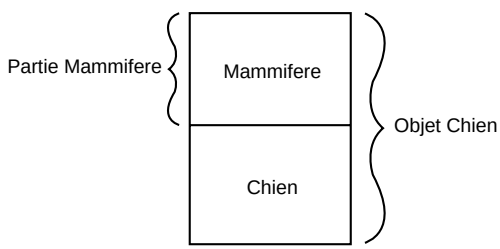


Figure 12.2
L'objet Chien après sa création.

Lorsque l'on crée une méthode virtuelle dans un objet, celui-ci doit garder la trace de cette fonction. Beaucoup de compilateurs créent une table de fonction virtuelle, nommée *v-table*. À chaque type correspond une table et chaque objet d'un type dispose d'un pointeur de table virtuel (appelé *vptr* ou v-pointeur), qui pointe sur cette table.

Bien que les implémentations puissent varier, tous les compilateurs doivent accomplir la même tâche.

Le v-pointeur de chaque objet pointe sur la v-table qui contient elle-même des pointeurs vers chaque méthode virtuelle (les pointeurs de fonctions sont traités au Chapitre 15). Lorsque la partie `Mammifere` de l'objet `Chien` est créée, le vptr est donc initialisé pour pointer sur la zone appropriée de la v-table, comme le montre la Figure 12.3.

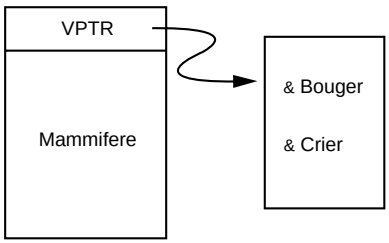


Figure 12.3
La v-table d'un objet Mammifere.

Lorsque le constructeur de `Chien` est appelé et que la partie `Chien` de cet objet est ajoutée, le v-pointeur est modifié pour pointer sur la fonction virtuelle redéfinie (le cas échéant) dans l'objet `Chien` (voir Figure 12.4).

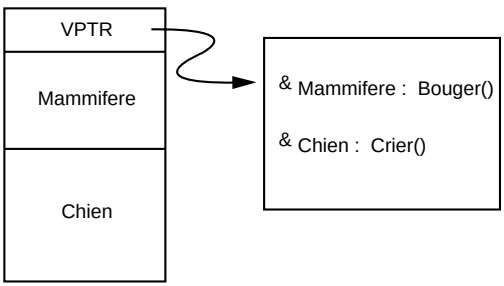


Figure 12.4
La v-table d'un objet Chien.

Lorsque l'on utilise un pointeur vers un objet `Mammifere`, le v-pointeur continue à pointer sur la fonction appropriée au type "réel" de l'objet. Ainsi, lorsqu'on va appeler `Crier()`, on obtiendra bien la fonction adéquate.

Accès aux méthodes à partir d'une classe de base

Nous avons vu des méthodes d'une classe dérivée, auxquelles on a accédé à partir d'une classe de base grâce au mécanisme des fonctions virtuelles. Que se passerait-il s'il y avait une méthode dans la classe dérivée qui ne soit pas dans la classe de base ? Pourriez-vous y accéder de la même manière qu'en utilisant la classe de base pour accéder aux méthodes virtuelles ? Il ne devrait pas y avoir de conflit de noms puisque seule la classe dérivée dispose de cette méthode.

Si l'objet `Chien` possède une méthode `RemuerQueue()` par exemple, qui ne figure pas dans la classe `Mammifere`, vous ne pouvez pas vous servir du pointeur de la classe de base pour avoir accès à la fonction de la classe dérivée. La méthode `RemuerQueue()` n'étant pas virtuelle et n'appartenant pas à la classe `Mammifere`, elle ne peut être appelée si vous n'êtes ni un `Chien` ni un pointeur sur celui-ci.

On pourrait transtyper le pointeur de `Mammifere` en pointeur sur un `Chien`. Toutefois, cette pratique est à déconseiller, car elle n'est pas sûre si le `Mammifere` pointé n'est pas un `Chien` : il existe un méthode bien plus fiable d'appeler `RemuerQueue()`. Les transtypages doivent être utilisés avec la plus grande prudence, car ils engendrent des erreurs d'exécution. Pour en savoir plus, reportez-vous aux Chapitres 15 et 20.

Découpage des données

La magie des méthodes virtuelles n'agit que sur les pointeurs et les références. Passer un objet par valeur ne permet pas d'appeler une fonction virtuelle, comme le montre le Listing 12.10.

Listing 12.10 : Passage de paramètres par valeur et découpage de données

```
 1:  // Listing 12.10 Passage de paramètres par valeur
1a:  // et découpage de données
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Mammifere
 6:  {
 7:  public:
 8:      Mammifere():sonAge(1) {  }
 9:      virtual ~Mammifere() { }
10:      virtual void Crier() const
10a:         { cout << "Le cri du mammifère !\n"; }
11:
12:  protected:
13:     int sonAge;
14:  };
15:
16:  class Chien : public Mammifere
17:  {
18:  public:
19:     void Crier()const { cout << "Ouah Ouah !\n"; }
20:  };
21:
22:  class Chat : public Mammifere
23:  {
24:  public:
25:     void Crier()const { cout << "Miaou !\n"; }
26:  };
27:
28:  void FoncVal(Mammifere);
29:  void FoncPtr(Mammifere*);
```

```
30:  void FoncRef(Mammifere&);
31:  int main()
32:  {
33:     Mammifere* ptr = 0;
34:     int choix;
35:     while (1)
36:     {
37:        bool fQuit = false;
38:        cout << "(1)Chien (2)Chat (0)Quitter - Votre choix : ";
39:        cin >> choix;
40:        switch (choix)
41:        {
42:        case 0: fQuit = true;
43:              break;
44:        case 1: ptr = new Chien;
45:              break;
46:        case 2: ptr = new Chat;
47:              break;
48:        default: ptr = new Mammifere;
49:              break;
50:        }
51:        if (fQuit)
52:           break;
53:        FoncPtr(ptr);
54:        FoncRef(*ptr);
55:        FoncVal(*ptr);
56:     }
57:     return 0;
58:  }
59:
60:  void FoncVal(Mammifere ValeurMammifere)
61:  {
62:     ValeurMammifere.Crier();
63:  }
64:
65:  void FoncPtr(Mammifere * pMammifere)
66:  {
67:     pMammifere->Crier();
68:  }
69:
70:  void FoncRef(Mammifere & rMammifere)
71:  {
72:     rMammifere.Crier();
73:  }
```

Ce programme produit le résultat suivant :

```
(1)Chien (2)Chat (0)Quitter - Votre choix : 1
Ouah Ouah !
Ouah Ouah !
Le cri du mammifère !

(1)Chien (2)Chat (0)Quitter - Votre choix : 2
Miaou !
Miaou !
Le cri du mammifère !

(1)Chien (2)Chat (0)Quitter - Votre choix : 0
```

Le programme commence par les déclarations des versions réduites de `Mammifere`, `Chien` et `Chat`. Les fonctions `FoncPtr()`, `FoncRef()` et `FoncVal()` sont ensuite déclarées et prennent respectivement un pointeur sur un objet `Mammifere`, une référence à un objet `Mammifere` et un objet `Mammifere`. Comme vous pouvez le constater aux lignes 60 à 73, elles appellent toutes les trois la méthode `Crier()`.

L'utilisateur a le choix entre un chien et un chat. En fonction de la valeur saisie, les lignes 44 à 46 créent un pointeur vers le type correspondant.

Sur la première ligne du résultat, l'utilisateur a choisi l'option 1. Un objet `Chien` est alors créé sur le tas (ligne 44), puis passé successivement aux trois fonctions comme pointeur en ligne 53, comme référence en ligne 54 et par valeur en ligne 55.

Le pointeur et les appels de référence permettent d'appeler la fonction virtuelle `Chien->Crier()` (voir les deux premières lignes du résultat après le choix de l'utilisateur).

Par contre, on a passé par valeur le pointeur déréférencé à la fonction définie aux lignes 60 à 63. Celle-ci attendant un objet `Mammifere`, le compilateur découpe la partie de base contenue dans l'objet `Chien` et ne conserve que la partie `Mammifère`. Lorsque l'on appelle la méthode `Crier()` de `Mammifere` à la ligne 72, seules les informations sur les mammifères sont donc disponibles, tout ce qui concerne les chiens a disparu. On peut le constater avec la troisième ligne du résultat, après le choix de l'utilisateur. Cet effet est appelé découpage, car les parties `Chien` (celles de votre classe dérivée) de votre objet ont été supprimées lors de la conversion en `Mammifere` (la classe de base).

Cette opération est ensuite répétée pour l'objet `Chat`, avec les mêmes conséquences.

Destructeurs virtuels

Il est courant et classique de passer un pointeur sur un objet d'une classe dérivée lorsque le programme attend un pointeur sur un objet de la classe de base. Que se passe-t-il si le pointeur de l'objet dérivé est supprimé ? Si le destructeur est virtuel, comme cela devrait

être le cas, tout se passera bien – c'est le destructeur de la classe dérivée qui sera appelé. Ce destructeur appelant automatiquement celui de la classe de base, tout l'objet sera correctement détruit.

La règle est la suivante : si l'une des méthodes d'une classe est virtuelle, le destructeur de cette classe doit l'être lui aussi.



Les listings de ce chapitre comprenaient des destructeurs virtuels. Vous savez maintenant pourquoi ! En général, il vaut toujours mieux que les destructeurs soient virtuels.

Constructeurs de copie virtuels

Les constructeurs ne pouvant pas être virtuels, on ne peut donc techniquement pas écrire un constructeur de copie virtuel. Cependant, un programme a parfois désespérément besoin de pouvoir passer un pointeur vers un objet de la classe de base et d'obtenir une copie de l'objet dérivé correspondant qui a été créé.

La solution la plus courante consiste à créer une méthode `Clone()` dans la classe de base en la déclarant virtuelle. Cette méthode crée une copie de l'objet de la classe courante et la renvoie.

Toutes les classes dérivées redéfinissant la méthode `Clone()`, la copie créée sera une copie de la classe dérivée. Cette procédure est mise en œuvre dans le Listing 12.11.

Listing 12.11 : Constructeur de copie virtuel

```
 1:  //Listing 12.11 Constructeur de copie virtuel
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Mammifere
 6:  {
 7:  public:
 8:      Mammifere():sonAge(1)
8a:         { cout << "Constructeur de Mammifere...\n"; }
 9:      virtual ~Mammifere()
9a:         { cout << "Destructeur de Mammifere...\n"; }
10:     Mammifere const Mammifere & rhs);
11:     virtual void Crier() const
11a:       { cout << "Le cri du mammifère !\n"; }
12:     virtual Mammifere* Clone()
12a:       { return new Mammifere(*this); }
```

```
13:     int GetAge()const { return sonAge; }
14:  protected:
15:     int wAge;
16:  };
17:
18:  Mammifere::Mammifere (const Mammifere & rhs):
18a    sonAge(rhs.GetAge())
19:  {
20:     cout << "Constructeur de copie de Mammifere...\n";
21:  }
22:
23:  class Chien : public Mammifere
24:  {
25:  public:
26:     Chien() { cout << "Constructeur de Chien...\n"; }
27:     virtual ~Chien() { cout << "Destructeur de Chien...\n"; }
28:     Chien(const Chien & rhs);
29:     void Crier()const { cout << "Ouah Ouah !\n"; }
30:     virtual Mammifere* Clone() { return new Chien(*this); }
31:  };
32:
33:  Chien::Chien(const Chien & rhs):
34:    Mammifere(rhs)
35:  {
36:     cout << "Constructeur de copie de Chien...\n";
37:  }
38:
39:  class Chat : public Mammifere
40:  {
41:  public:
42:     Chat() { cout << "Constructeur de Chat...\n"; }
43:     ~Chat() { cout << "Destructeur de Chat...\n"; }
44:     Chat(const Chat &);
45:     void Crier()const { cout << "Miaou !\n"; }
46:     virtual Mammifere* Clone() { return new Chat(*this); }
47:  };
48:
49:  Chat::Chat(const Chat & rhs):
50:    Mammifere(rhs)
51:  {
52:     cout << "Constructeur de copie de Chat...\n";
53:  }
54:
55:  enum ANIMALS { MAMMIFERE, CHIEN, CHAT};
56:  const int NbreTypesAnimaux = 3;
57:  int main()
58:  {
```

```
59:     Mammifere *Tableau[NbreTypesAnimaux ];
60:     Mammifere* ptr;
61:     int choix, i;
62:     for ( i = 0; i < NbreTypesAnimaux; i++)
63:     {
64:        cout <<
64a:          "(1)Chien (2)Chat (3)Mammifère - Votre choix : ";
65:        cin >> choix;
66:        switch (choix)
67:        {
68:        case CHIEN:   ptr = new Chien;
69:                 break;
70:        case CHAT:   ptr = new Chat;
71:                 break;
72:        default:   ptr = new Mammifere;
73:                 break;
74:        }
75:        Tableau[i] = ptr;
76:     }
77:     Mammifere *AutreTableau[NbreTypesAnimaux];
78:     for (i = 0; i < NbreTypesAnimaux; i++)
79:     {
80:        Tableau[i]->Crier();
81:        AutreTableau[i] = Tableau[i]->Clone();
82:     }
83:     for (i = 0; i < NbreTypesAnimaux; i++)
84:        AutreTableau[i]->Crier();
85:     return 0;
86:  }
```

Ce programme produit le résultat suivant :

```
 1: (1)Chien (2)Chat (3)Mammifère - Votre choix  : 1
 2: Constructeur de Mammifere...
 3: Constructeur de Chien...
 4: (1)Chien (2)Chat (3)Mammifère - Votre choix  : 2
 5: Constructeur de Mammifere...
 6: Constructeur de Chat...
 7: (1)Chien (2)Chat (3)Mammifère - Votre choix : 3
 8: Constructeur de Mammifere...
 9: Ouah Ouah !
10: Constructeur de copie de Mammifere...
11: Constructeur de copie de Chien...
12: Miaou !
13: Constructeur de copie de Mammifere...
14: Constructeur de copie de Chat...
15: Le cri du mammifère !
```

```
16: Constructeur de copie de Mammifere...
17: Ouah Ouah !
18: Miaou !
19: Le cri du mammifère !
```

Le Listing 12.11 ressemble beaucoup aux deux précédents, sauf que la ligne 12 ajoute à la classe `Mammifere` une nouvelle méthode virtuelle appelée `Clone()`. Cette méthode renvoie un pointeur sur un nouvel objet `Mammifere`. Pour cela, elle appelle le constructeur de copie en passant en paramètre l'objet lui-même (`*this`) comme référence constante.

`Chien` et `Chat` redéfinissent toutes les deux la méthode `Clone()` en initialisant leurs données et en passant des copies de leurs objets à leurs propres constructeurs de copie. Cette méthode étant virtuelle, cela revient donc à créer un constructeur de copie "virtuel". Vous pouvez le constater lorsque la ligne 81 s'exécute.

L'utilisateur a le choix entre chiens, chats et mammifères. Les objets correspondants sont créés de la ligne 68 à la ligne 73. Pour chaque choix, un pointeur est stocké dans un tableau (ligne 75).

Lorsque le programme boucle sur le tableau (lignes 78 à 82), chaque objet appelle ses méthodes `Crier()` et `Clone()`. Le résultat de l'appel à `Clone()` à la ligne 81 est un pointeur vers une copie de l'objet, qui est ensuite stocké dans un second tableau.

À la première ligne du résultat, l'utilisateur a choisi l'option 1 pour créer un objet `Chien`. Les constructeurs de `Mammifere` et de `Chien` sont alors appelés. La procédure est semblable lorsqu'il choisit un `Chat` (ligne 4) et un `Mammifere` (ligne 8).

La ligne 9 du résultat est le résultat de l'appel à `Crier()` sur le premier objet, le `Chien`. La méthode virtuelle `Crier()` est appelée et invoque donc la version correcte de cette méthode. Puis, on appelle la méthode `Clone()` et, puisqu'elle est également virtuelle, c'est la méthode `Clone()` de `Chien` qui est invoquée. Le constructeur de `Mammifere` et le constructeur de copie de `Chien` sont donc appelés.

Le même traitement s'exécute pour le chat et le mammifère (lignes 12 à 14 et lignes 15 et 16) Pour finir, le nouveau tableau est parcouru (lignes 83 et 84) et chaque élément appelle sa méthode `Crier()` (lignes 17 à 19 du résultat).

Avantages et inconvénients des méthodes virtuelles

Les objets qui possèdent des méthodes virtuelles consomment plus de mémoire à cause de la v-table qu'ils doivent gérer. Si votre est très petite et que vous ne prévoyez pas que d'autres classes en hériteront, il est inutile d'avoir recours aux méthodes virtuelles.

La v-table occupe sa place en mémoire dès la création de la première fonction virtuelle et sa taille évolue peu par la suite (même si chaque nouvelle entrée augmente un peu sa

taille). Vous avez donc payé l'essentiel du billet d'entrée et vous pouvez alors en profiter : le destructeur devra être virtuel et toutes les autres méthodes devront sûrement l'être aussi. Prenez le temps d'examiner chacune des méthodes non virtuelles et assurez-vous d'avoir bien compris pourquoi elles ne le sont pas.

Faire	*Ne pas faire*
● Utiliser des méthodes virtuelles si vous envisagez de créer des classes dérivées. ● Utiliser un destructeur virtuel si la classe comporte au moins une méthode virtuelle.	● Déclarer un constructeur virtuel. ● Tenter d'accéder à des données privées dans une classe de base à partir d'une classe dérivée.

Questions-réponses

Q Les membres et les méthodes hérités sont-ils transmis aux générations suivantes ? Si `Chien` dérive de `Mammifere` et que cette dernière classe dérive elle-même de `Animal`, `Chien` hérite-t-il des méthodes et des données de `Animal` ?

R Oui. Les classes dérivées héritent de l'ensemble des méthodes et des données de toutes leurs classes de base, mais elles ne peuvent accéder qu'à celles qui sont protégées ou publiques.

Q Si, dans l'exemple précédent, `Mammifere` redéfinit une méthode de la classe `Animal`, la classe `Chien` accèdera-t-elle à la fonction d'origine ou à la fonction redéfinie ?

R Si `Chien` hérite de `Mammifere`, c'est la fonction redéfinie qui sera appelée.

Q Est-ce qu'une classe dérivée peut rendre privée une méthode publique de la classe de base ?

R Oui, la classe dérivée peut redéfinir la méthode et la rendre privée. Elle le demeure ensuite pour toutes les classes dérivées suivantes. Cela doit toutefois être évité autant que possible, car les utilisateurs de votre classe s'attendront ce qu'elle contienne la somme des méthodes fournies par ses ancêtres.

Q Pourquoi toutes les fonctions de classe ne sont-elles pas virtuelles ?

R La première fonction créée dans la table virtuelle consomme davantage de ressources que les autres, mais ce surcoût augmente peu ensuite. De nombreux programmeurs C++ pensent que si une méthode est virtuelle, toutes les autres doivent l'être également. D'autres ne sont pas d'accord et pensent qu'il doit toujours exister une bonne raison de le faire.

Q Supposons que la fonction `Fonc()` soit virtuelle dans une classe de base et surchargée pour prendre en paramètre un ou deux entiers. La classe dérivée redéfinit la fonction qui n'a qu'un paramètre. Laquelle des deux fonctions sera appelée lorsqu'un pointeur vers un objet dérivé invoquera la fonction avec deux paramètres ?

R La redéfinition de la fonction avec un seul paramètre a masqué les deux fonctions de la classe de base, ce qui les rend inutilisables. Le compilateur va produire une erreur indiquant que la fonction n'attend qu'un seul paramètre de type entier.

Testez vos connaissances

1. Qu'est-ce qu'une v-table ?
2. Qu'est-ce qu'un destructeur virtuel ?
3. Comment déclare-t-on un constructeur virtuel ?
4. Comment crée-t-on un constructeur de copie virtuel ?
5. Comment peut-on appeler une fonction membre d'une classe de base à partir d'une classe dérivée dans laquelle elle a été redéfinie ?
6. Comment peut-on appeler une fonction membre d'une classe de base à partir d'une classe dérivée dans laquelle elle n'a pas été redéfinie ?
7. Si une classe de base déclare une fonction virtuelle et que le mot-clé `virtual` est omis dans la classe dérivée lors de la redéfinition de cette fonction, sera-t-elle toujours virtuelle lorsqu'elle sera héritée par une classe de troisième génération ?
8. À quoi sert le mot-clé `protected` ?

Exercices

1. Déclarez une fonction virtuelle qui prend un entier en paramètre et renvoie `void`.
2. Déclarez une classe `Carre`, dérivée de `Rectangle`, elle-même dérivée de `Forme`.
3. Dans l'Exercice 2, le constructeur de `Forme` ne prend pas de paramètres, alors que celui de `Rectangle` attend une longueur et une largeur. Celui de `Carre` attend seulement une longueur. Montrez l'initialisation d'un `Carre` réalisée par son constructeur.
4. Écrivez un constructeur de copie virtuel pour la classe `Carre` de l'Exercice 3.
5. CHERCHEZ L'ERREUR dans cet extrait de programme :

```
void Fonction(Forme);
```

```
Forme * pRect = new Rectangle;
Fonction(*pRect);
```

6. CHERCHEZ L'ERREUR dans cet extrait de programme :

```
class Forme()
{
public:
     Forme();
     virtual ~Forme();
     virtual Forme(const Forme&);
};
```



13

Tableaux et chaînes

Au sommaire de ce chapitre

- Nature, rôle et déclaration des tableaux
- Nature des chaînes et création à partir de tableaux de caractères
- Relations entre tableaux et pointeurs
- Utilisation de l'arithmétique des pointeurs
- Les listes chaînées

Qu'est-ce qu'un tableau ?

Un tableau est une collection séquentielle d'emplacements en mémoire pouvant contenir, chacun, des données de même type. Chaque emplacement est appelé *élément* du tableau.

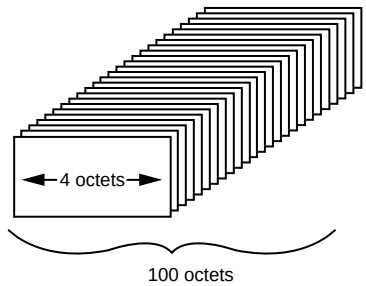
Pour déclarer un tableau, vous devez indiquer un type, suivi du nom du tableau et d'un nombre entre crochets, correspondant au nombre d'éléments du tableau. Par exemple :

```
long TableauLong[25];
```

Cette ligne déclare un tableau de 25 entiers longs, nommé `TableauLong`. Le compilateur réservera un espace suffisant en mémoire pour contenir ces 25 éléments. Si un entier long

occupe 4 octets, cette déclaration réserve donc 100 octets contigus en mémoire vive (voir Figure 13.1).

Figure 13.1
Déclaration d'un tableau.



Accès aux éléments de tableau

On accède à un élément particulier d'un tableau en indiquant son décalage par rapport au début du tableau. Les décalages d'éléments (appelés plus communément *indices*) étant numérotés à partir de 0, le premier élément du tableau `TableauLong` est représenté par `TableauLong[0]`, le second par `TableauLong[1]`, etc.

Cette syntaxe peut prêter à confusion. La déclaration `Tableau[3]`, par exemple, indique que `Tableau` contiendra trois éléments : `Tableau[0]`, `Tableau[1]` et `Tableau[2]`. Plus généralement, la déclaration `Tableau[n]` crée un tableau de `n` éléments numérotés de `Tableau[0]` à `Tableau[n-1]`. Ceci est dû au fait que l'indice est un décalage et que le premier élément est donc à 0 emplacement par rapport au début du tableau, le deuxième à un emplacement, et ainsi de suite.

En conséquence, `TableauLong[25]` est numéroté de `TableauLong[0]` à `TableauLong[24]`. Le Listing 13.1 déclare un tableau de cinq entiers, puis affecte une valeur à chacun de ses éléments.



À partir de maintenant, les numéros de ligne commenceront par zéro pour vous aider à vous souvenir que les tableaux C++ commencent aussi par zéro !

Listing 13.1 : Utilisation d'un tableau d'entiers

```
0:  //Listing 13.1 - Tableaux
1:  #include <iostream>
2:
3:  int main()
4:  {
```

```
 5:     int monTableau[5];   // Tableau de cinq entiers
 6:     int i;
 7:     for ( i = 0; i < 5; i++)  // 0 à 4
 8:     {
 9:         std::cout << "Valeur de monTableau[" << i << "] : ";
10:         std::cin >> monTableau[i];
11:     }
12:     for (i = 0; i < 5; i++)
13:        std::cout << i << ": " << monTableau[i] << std::endl;
14:     return 0;
15:  }
```

Ce programme produit le résultat suivant :

```
Valeur de monTableau[0] :  3
Valeur de monTableau[1] :  6
Valeur de monTableau[2] :  9
Valeur de monTableau[3] :  12
Valeur de monTableau[4] :  15
0 : 3
1 : 6
2 : 9
3 : 12
4 : 15
```

Le Listing 13.1 crée un tableau, vous fait saisir les valeurs de chaque élément, puis les affiche à l'écran. La déclaration de la ligne 5 indique que `monTableau` est un tableau d'entiers ; elle contient le chiffre 5 entre crochets, ce qui signifie que le tableau contient cinq éléments entiers, chacun pouvant être considéré comme une variable entière. À la ligne 7, la variable de boucle s'incrémente de 0 à 4 afin d'initialiser les éléments du tableau avec la valeur saisie par l'utilisateur.

Si l'on se penche sur la ligne 10, on voit que l'accès à chaque élément se fait à l'aide du nom du tableau suivi de crochets, contenant le décalage. Chaque élément peut ensuite être traité comme une variable du type des éléments du tableau.

La première valeur est stockée dans `monTableau[0]`, la deuxième dans `monTableau[1]`, etc. La deuxième boucle permet d'afficher toutes les valeurs à l'écran.



La numérotation des éléments d'un tableau commence à 0 (jamais à 1), ce qui est une cause fréquente d'erreurs pour les débutants en C++. L'indice doit être considéré comme un décalage. Par exemple, un tableau contenant 10 éléments sera numéroté de `Tableau[0]` à `Tableau[9]`, l'élément `Tableau[10]` correspondant à un onzième élément situé hors limite.

Écriture d'éléments hors limite

Lorsque l'on écrit une valeur dans un tableau, le compilateur calcule l'emplacement de stockage en fonction de la taille de chaque élément et son indice. Supposons, par exemple, que vous souhaitiez stocker une valeur dans `TableauLong[5]` alors que le tableau ne comprend que 5 éléments. Le compilateur va multiplier la valeur du décalage (5) par la taille de chaque élément (4 octets, ici) et donc déplacer le pointeur de 20 octets par rapport au début du tableau. Enfin, il va écrire la nouvelle valeur à cet emplacement.

En vérité, le compilateur réalise la copie de la valeur à l'endroit indiqué, même si ce dernier se situe en dehors de la limite supérieure du tableau. Par exemple, si vous tentez de stocker une valeur dans `TableauLong[50]`, le compilateur se placera à 200 octets à partir du début du tableau, puis écrasera les données qui se trouvent à cet emplacement en les remplaçant par la nouvelle valeur. Cette opération peut, dans le meilleur des cas, produire une erreur immédiate (dans le meilleur des cas). Dans le pire des cas, cette erreur ne surviendra pas tout de suite et vous finirez par obtenir des résultats bizarres que vous aurez beaucoup de mal à expliquer.

Le compilateur se comporte comme un aveugle estimant une distance à partir d'une maison, `Maison[0]`. Si vous lui demandez d'aller à la sixième maison de la rue, il se dira "je dois aller 5 maisons plus loin. Chaque maison faisant quatre grands pas de large, je dois avancer de 20 pas.". Si vous lui demandez d'aller à `Maison[100]` alors que la rue n'a que 25 maisons, il se déplacera de 400 pas et se retrouvera sûrement sous les roues d'un camion avant de les avoir tous parcourus. Vous devez donc être sûr de l'endroit où vous l'envoyez.

Le Listing 13.2 écrit une valeur après la limite supérieure d'un tableau. Compilez ce listing pour voir les messages d'erreur et d'avertissement qui s'affichent. S'il n'y en a pas, faites d'autant plus attention lorsque vous travaillerez avec les tableaux !



Ce programme contient des erreurs classiques à ne pas commettre. Ne l'exécutez pas, car il risquerait de faire planter votre système.

Listing 13.2 : Stockage d'une valeur au-delà de la limite supérieure d'un tableau

```
0:  // Listing 13.2 - Écriture d'une valeur au-delà de la limite
1:  // supérieure d'un tableau
2:  #include <iostream>
3:  using namespace std;
4:
5:  int main()
6:  {
```

```
 7:     // sentinelles
 8:     long SentinelUne[3];
 9:     long TableauCible[25]; // Tableau à remplir
10:     long SentinelDeux[3];
11:     int i;
12:     for (i = 0; i < 3; i++)
13:     {
14:        SentinelUne[i] = 0;
15:        SentinelDeux[i] = 0;
16:     }
17:     for (i = 0; i < 25; i++)
18:        TableauCible[i] = 10;
19:
19a:    // test de la valeur actuelle (devrait être 0)
20:     cout << "Test 1 : \n";
21:     cout << "TableauCible[0] : " << TableauCible[0] << endl;
22:     cout << "TableauCible[24] : " << TableauCible[24]
22a:         << endl << endl;
23:
24:     for (i = 0; i < 3; i++)
25:     {
26:        cout << "SentinelUne[" << i << "] : ";
27:        cout << SentinelUne[i] << endl;
28:        cout << "SentinelDeux[" << i << "] : ";
29:        cout << SentinelDeux[i]<< endl;
30:     }
31:
32:     cout << "\Affectation en cours...";
33:     for (i = 0; i <= 25; i++)    // Va un peu trop loin !
34:        TableauCible[i] = 20;
35:
36:     cout << "\nTest 2 : \n";
37:     cout << "TableauCible[0] : " << TableauCible[0] << endl;
38:     cout << "TableauCible[24] : " << TableauCible[24] << endl;
39:     cout << "TableauCible[25] : " << TableauCible[25]
39a:             << endl << endl;
40:     for (i = 0; i < 3; i++)
41:     {
42:        cout << "SentinelUne[" << i << "] : ";
43:        cout << SentinelUne[i]<< endl;
44:        cout << "SentinelDeux[" << i << "] : ";
45:        cout << SentinelDeux[i]<< endl;
46:     }
47:
48:     return 0;
49: }
```

Ce programme produirait le résultat suivant :

```
Test 1 :
TableauCible[0] : 10
TableauCible[24] : 10

SentinelUne[0] : 0
SentinelDeux[0] : 0
SentinelUne[1] : 0
SentinelDeux[1] : 0
SentinelUne[2] : 0
SentinelDeux[2] : 0

Affectation en cours...
Test 2 :
TableauCible[0] :  20
TableauCible[24] : 20
TableauCible[25] : 20

SentinelUne[0] : 20
SentinelDeux[0] : 0
SentinelUne[1] : 0
SentinelDeux[1] : 0
SentinelUne[2] : 0
SentinelDeux[2] : 0
```

Les deux tableaux de trois entiers longs déclarés aux lignes 8 et 10 vont jouer le rôle de sentinelles autour du tableau `TableauCible`. Ces sentinelles sont initialisées à 0 par les lignes 12 à 16. Comme elles sont déclarées avant et après le tableau cible, il y a de fortes chances qu'elles soient placées en mémoire de part et d'autre de ce dernier. Si le programme écrit des données après la fin de `TableauCible`, les sentinelles seront donc probablement modifiées. Certains compilateurs allouent vers le bas de la mémoire alors que d'autres allouent vers le haut, c'est pour cette raison que l'on a encadré le tableau cible par deux sentinelles.

Lors du premier test, réalisé de la ligne 20 à la ligne 30, les valeurs des sentinelles sont correctes, comme leur affichage, tout comme le premier et le dernier élément de `TableauCible`. À la ligne 34, les membres de `TableauCible` sont tous réaffectés avec la valeur 20, mais le compteur atteint (ligne 34) la valeur de décalage 25, qui n'existe pas le tableau cible.

Les valeurs de `TableauCible` apparaissent à l'écran (lignes 37 à 39) sous forme de deuxième test. Vous pouvez constater (ligne 39) que `TableauCible[25]` contient bien le nombre 20. Cependant, lorsque les sentinelles sont affichées, on constate que la valeur de `SentinelUne[0]` a été modifiée. En effet, la zone mémoire située à 25 éléments de

`TableauCible[0]` est la même que `SentinelUne[0]`. Lorsqu'on a tenté d'accéder à la valeur inexistante `TableauCible[25]`, on a donc accédé en réalité au contenu de `SentinelUne[0]`.



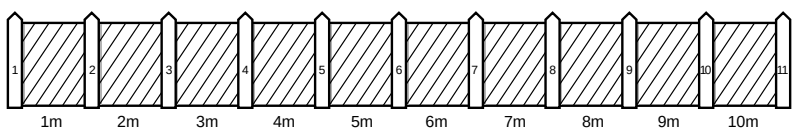
Les compilateurs utilisant différemment la mémoire, vos résultats peuvent varier. Il se peut que les sentinelles ne soient pas écrasées. Dans ce cas, essayez de modifier la ligne 33 pour utiliser un autre indice (passez de 25 à 26). Ceci augmentera la probabilité d'écraser une sentinelle, mais vous risquez bien sûr d'écraser autre chose ou de faire planter votre système.

Cette erreur est difficile à détecter, car la valeur de `SentinelUne[0]` est modifiée dans une partie du programme qui n'accède pas directement à `SentinelUne`.

Erreurs d'intervalle

Il est si courant d'écrire une valeur après la limite supérieure d'un tableau que cette erreur a son propre nom : on l'appelle erreur d'intervalle ou erreur de borne. Lorsque l'on demande combien il y a de piquets sur une barrière de 10 mètres, sachant que l'espacement entre les piquets est de un mètre, la plupart des gens répondent sans réfléchir : "10 !", or la bonne réponse est 11 (voir Figure 13.2).

Figure 13.2
Erreurs d'intervalle.



Cette façon de compter "en partant par défaut de un" est la bête noire des programmeurs C++ débutants. Avec le temps, vous vous habituerez à l'idée qu'un tableau de 25 éléments est numéroté de 0 à 24.



Certains programmeurs désignent `Tableau[0]` comme l'élément 0. Cette habitude est à proscrire, car si `Tableau[0]` est l'élément 0, à quoi correspond `Tableau[1]` ? au premier élément ? Dans ce cas, lorsque vous verrez `Tableau[24]`, réaliserez-vous qu'il ne s'agit pas du vingt-quatrième élément du tableau, mais du vingt-cinquième ? Pour éviter toute confusion, il est préférable de dire que `Tableau[0]` est au décalage 0 et correspond donc au premier élément.

Initialisation des tableaux

Lors de la déclaration d'un tableau d'éléments de type prédéfini (entiers, caractères, par exemple), vous pouvez en profiter pour initialiser ses valeurs. Pour ce faire, vous devez placer le signe égal (=) après la déclaration du tableau, suivi d'une liste de valeurs entre accolades, séparées les unes des autres par des virgules. Exemple :

```
int TableauInt[5] = { 10, 20, 30, 40, 50 };
```

Cette instruction déclare `TableauInt` comme un tableau de cinq entiers et affecte la valeur 10 à `TableauInt[0]`, la valeur 20 à `TableauInt[1]`, etc.

Si vous omettez de préciser la taille du tableau, le compilateur produira un tableau contenant autant d'éléments que la liste et leur affectera les valeurs respectives. Par exemple, l'expression :

```
int TableauInt[] = { 10, 20, 30, 40, 50 };
```

produit le même tableau que précédemment, un tableau de cinq éléments.

Il est interdit d'initialiser plus d'éléments que n'en contient le tableau. Exemple :

```
int TableauInt[5] = { 10, 20, 30, 40, 50, 60 };
```

Cette expression produira une erreur de compilation car vous avez tenté d'affecter six valeurs aux cinq éléments du tableau. En revanche, vous avez le droit d'écrire :

```
int TableauInt[5] = { 10, 20 };
```

Ici, vous avez déclaré un tableau de cinq éléments et initialisé uniquement les deux premiers, `TableauInt[0]` et `TableauInt[1]`.

Faire	*Ne pas faire*
● Laisser le compilateur calculer la taille des tableaux initialisés. ● Se rappeler que le premier élément d'un tableau porte le numéro de décalage zéro.	● Écrire au-delà de la limite supérieure du tableau. ● Donner des noms confus aux tableaux. Les noms doivent être évocateurs, exactement comme pour toute autre variable.

Déclaration de tableaux

Pour l'instant, nous avons utilisé des "nombres magiques" : 3 pour la taille des tableaux de sentinelles et 25 pour la taille de `TableauCible`. Il vaut mieux utiliser des constantes pour pouvoir modifier toutes ces valeurs à un seul endroit.

Un tableau peut porter un nom de variable, à condition qu'il ne s'agisse pas du nom d'une variable ou d'un tableau existant dans la même portée. Vous ne pouvez donc pas avoir à la fois un tableau `Chats[5]` et une variable `Chats`.

En outre, lors de la déclaration du nombre d'éléments, vous pouvez utiliser une constante ou une énumération. C'est même préférable à des nombres littéraux car cela permet de regrouper au même endroit le contrôle du nombre des éléments. Le Listing 13.2 utilisait des nombres littéraux : si vous souhaitez changer le `TableauCible` pour qu'il ne contienne plus que vingt éléments, vous deviez modifier plusieurs lignes de code. Si vous aviez utilisé une constante, il suffisait de ne modifier que la valeur de cette constante.

Indiquer le nombre d'éléments, ou la dimension, avec une énumération est un peu différent. Le Listing 13.3 montre la création d'un tableau contenant une valeur par jour de la semaine.

Listing 13.3 : Utilisation de constantes et d'énumérations dans les tableaux

```
0:  // Listing 13.3
1:  // Dimensionnement de tableaux
2:  // à l'aide de constantes et d'énumérations
3:  #include <iostream>
4:  int main()
5:  {
6:      enum Semaine { Dim, Lun, Mar,
7:              Mer, Jeu, Ven, Sam, NbJoursSemaine };
8:      int TabSemaine[NbJoursSemaine] =
8a:             { 10, 20, 30, 40, 50, 60, 70 };
9:
10:     std::cout << "La valeur de Mardi est : "
10a               << TabSemaine[Mar];
11:     return 0;
12:  }
```

Ce programme produit le résultat suivant :

```
La valeur de Mardi est : 30
```

La ligne 6 crée l'énumération `Semaine`, qui contient huit membres. Le premier élément (`Dim`) et le dernier élément (`NbJoursSemaine`) correspondent respectivement à 0 et à 7. La ligne 8 déclare le tableau `TabSemaine` de `NbJoursSemaines` éléments, c'est-à-dire 7.

La ligne 10 utilise la constante énumérée `Mar` comme indice du tableau. Cette constante correspondant à 2, l'élément désigné est donc le troisième tableau, `TabSemaine`[2]. La valeur obtenue est affichée à la ligne 10.

Tableaux

Pour déclarer un tableau, indiquez le type des objets à stocker, suivi du nom du tableau et de la valeur d'indice correspondant au nombre d'éléments.

Exemple 1

```
int TableauInt[90];
```

Exemple 2

```
long * TabPointeursSurLongs[100];
```

Pour accéder aux éléments du tableau, indiquez l'indice souhaité.

Exemple 1

```
// affecter le 9e élément de TableauInt à neuviemeElt
int neuviemeElt = TableauInt[8];
```

Exemple 2

```
// affecter le 9e élément de TabPtrSurLongs à pLong
long * pLong = TabPtrSurLongs[8];
```

Un tableau de *n* éléments est toujours numéroté de 0 à *n-1*.

Tableaux d'objets

N'importe quel objet, d'un type prédéfini ou non, peut être stocké dans un tableau. Il suffit d'indiquer au compilateur le type des éléments à stocker et leur nombre : le compilateur pourra alors calculer l'espace mémoire nécessaire pour chaque élément en fonction de la déclaration de la classe. La classe des éléments d'un tableau doit comprendre un constructeur par défaut sans paramètre, afin que les objets puissent être créés au moment de la définition du tableau.

L'accès aux données membres d'un tableau d'objets se fait en deux étapes. On identifie d'abord l'élément du tableau à l'aide de l'opérateur d'indexation (`[ ]`), puis on lui ajoute l'opérateur d'accès aux membres (`.`). Le Listing 13.4 montre comment créer et manipuler un tableau de cinq objets `Chat`.

Listing 13.4 : Création d'un tableau d'objets

```
0:  // Listing 13.4 - Un tableau d'objets
1:
2:  #include <iostream>
3:  using namespace std;
```

```
 4:
 5:  class Chat
 6:  {
 7:  public:
 8:      Chat() { sonAge = 1; sonPoids = 5; }
 9:      ~Chat() {}
10:     int GetAge() const { return sonAge; }
11:     int GetPoids() const { return sonPoids; }
12:     void SetAge(int age) { sonAge = age; }
13:
14:  private:
15:     int sonAge;
16:     int sonPoids;
17:  };
18:
19:  int main()
20:  {
21:     Chat Portee[5];
22:     int i;
23:    for (i = 0; i < 5; i++)
24:        Portee[i].SetAge(2*i +1);
25:
26:     for (i = 0; i < 5; i++)
27:     {
28:        cout << "Chat numéro " << i+1 << " : ";
29:        cout << Portee[i].GetAge() << endl;
30:     }
31:     return 0;
32:  }
```

Ce programme produit le résultat suivant :

```
Chat numéro 1 : 1
Chat numéro 2 : 3
Chat numéro 3 : 5
Chat numéro 4 : 7
Chat numéro 5 : 9
```

La classe `Chat` est déclarée de la ligne 5 à la ligne 17. Elle doit posséder un constructeur par défaut pour que ses objets puissent être créés dans un tableau. Ici, ce constructeur par défaut est déclaré et défini à la ligne 8. Pour chaque `Chat`, l'âge par défaut est fixé à 1 et le poids par défaut à 5. N'oubliez pas que si vous définissez un autre constructeur, le constructeur par défaut ne sera pas produit par le compilateur (ce qui vous oblige à le créer vous-même).

La première boucle `for` fixe l'âge des cinq objets du tableau (lignes 23 et 24), la seconde (lignes 26 à 30) accède à chaque élément et appelle sa méthode `GetAge()`, ce qui permet d'afficher l'âge de tous les chats.

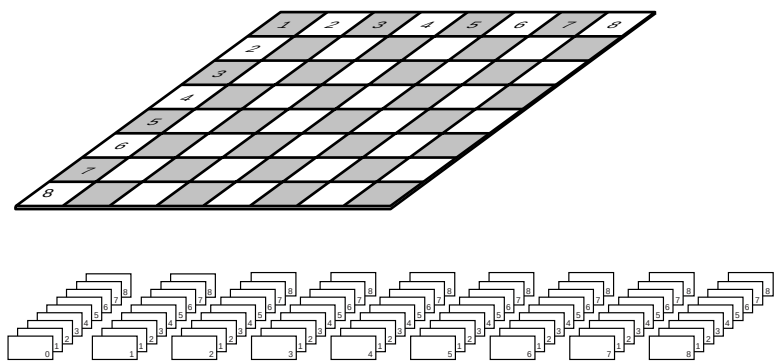
La méthode `GetAge()` de chaque chat est appelée en accédant à chacun des éléments du tableau `Portee`, suivi de l'opérateur point (`.`) et du nom de la fonction membre. Vous pouvez accéder à d'autres membres et méthodes de la même manière.

Tableaux multidimensionnels

Un tableau peut avoir plus d'une dimension. Chacune d'elles est représentée par un indice du tableau : un tableau bidimensionnel aura donc deux indices, un tableau tridimensionnel trois indices, et ainsi de suite. Vous pouvez créer autant de dimensions que vous le souhaitez, bien qu'en pratique la plupart de vos tableaux n'auront qu'une ou deux dimensions.

Un échiquier est un bon exemple de tableau à deux dimensions. Une dimension représente les 8 lignes, l'autre les 8 colonnes (voir Figure 13.3).

Figure 13.3
L'échiquier est un tableau bidimensionnel.



Supposons que vous ayez une classe `CARRE`. La déclaration d'un tableau `Echiquier` serait alors :

```
CARRE Echiquier[8][8];
```

Vous pourriez également représenter cet échiquier comme un tableau à une dimension de 64 carrés :

```
CARRE Echiquier[64];
```

Cependant, ce tableau ne correspondrait pas autant à un véritable échiquier qu'un tableau à deux dimensions. Lorsque la partie commence, le roi se situe à la quatrième position de

la première ligne. La numérotation commençant à zéro, cette position s'exprime donc ainsi :

```
Echiquier[0][3];
```

en supposant que le premier indice correspond aux lignes et le second aux colonnes.

Initialisation de tableaux à plusieurs dimensions

Comme les tableaux à une dimension, les tableaux à plusieurs dimensions peuvent être initialisés. Les valeurs sont affectées dans l'ordre des éléments, le dernier indice (celui qui est le plus à droite) évoluant pendant que le précédent reste stable (comme un compteur kilométrique). Si vous avez la déclaration suivante :

```
int Tableau[5][3];
```

les trois premiers éléments iront donc dans `Tableau[0]`, les trois suivants dans `Tableau[1]`, etc. :

```
int Tableau[5][3] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 };
```

Pour plus de clarté, vous pouvez regrouper les différentes initialisations entre accolades. Exemple :

```
int Tableau[5][3] = {  {1,2,3},
                       {4,5,6},
                       {7,8,9},
                       {10,11,12},
                       {13,14,15} };
```

Le compilateur ne tient pas compte des accolades internes, mais elles clarifient la répartition des nombres.

Lorsque vous initialisez les éléments d'un tableau, chaque valeur doit être séparée de la suivante par une virgule, indépendamment des accolades. L'initialisation complète doit figurer entre accolades et se terminer par un point-virgule.

Le Listing 13.5 crée un tableau à deux dimensions, dont la première est un ensemble de nombres de 0 à 4. La seconde regroupe les doubles des valeurs de la première.

Listing 13.5 : Création d'un tableau à plusieurs dimensions

```
0:  // Listing 13.5 - Création d'un tableau à plusieurs dimensions
1:  #include <iostream>
2:  using namespace std;
```

```
 3:
 4:  int main()
 5:  {
 6:      int UnTableau[2][5] =
6a:             { {0,1,2,3,4}, {0,2,4,6,8}};
 7:      for (int i = 0; i < 2; i++)
 8:      {
 9:        for (int j = 0; j < 5; j++)
10:        {
11:           cout << "UnTableau[" << i << "][" << j << "] : ";
12:           cout << UnTableau[i][j]<< endl;
13:        }
14:     }
15:     return 0;
16:  }
```

Ce programme produit le résultat suivant :

```
UnTableau[0][0] : 0
UnTableau[0][1] : 1
UnTableau[0][2] : 2
UnTableau[0][3] : 3
UnTableau[0][4] : 4
UnTableau[1][0] : 0
UnTableau[1][1] : 2
UnTableau[1][2] : 4
UnTableau[1][3] : 6
UnTableau[1][4] : 8
```

La ligne 6 déclare `UnTableau` comme un tableau à deux dimensions. La première dimension indique qu'il y aura deux ensembles de données et la deuxième est formée de cinq entiers. Cela crée donc une matrice de 2 lignes par 5 colonnes, comme le montre la Figure 13.4

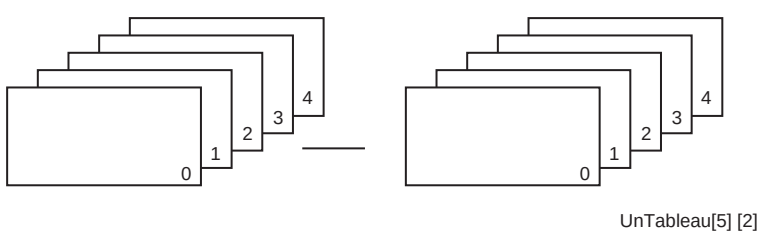


Figure 13.4
Tableau de 2 par 5.

Les valeurs sont initialisées sur la base des deux ensembles de nombres. Le premier comprend les nombres initiaux, le deuxième les nombres doublés. Dans ce listing, les valeurs sont simplement définies, mais elles pourraient être calculées. Les lignes 7 et 9

créent deux boucles `for` imbriquées. La boucle externe (à partir de la ligne 7) traite chaque membre de la première dimension (chacun des deux ensembles), puis appelle la boucle interne (à partir de la ligne 9) qui traite chaque élément de la seconde dimension, comme le montrent les valeurs affichées à l'écran. `UnTableau[0][0]` est donc suivi de `UnTableau[0][1]`. L'indice de la première dimension est incrémenté uniquement après l'incrémentation de l'indice de la seconde dimension. Le traitement recommence alors.

Tableaux et mémoire

La déclaration d'un tableau indique explicitement le nombre d'éléments qui le composent. Le compilateur se charge alors de réserver un espace suffisant en mémoire vive, même si vous n'en faites pas usage. Cette opération ne pose aucun problème lorsque vous savez exactement combien le tableau contiendra d'objets. Par exemple, un échiquier se compose de 64 cases, alors qu'une portée comprend entre un et dix matous. Si vous n'avez aucune idée du nombre d'éléments que devra contenir un tableau, vous devez faire appel à des structures de données plus complexes.

Cet ouvrage traite des tableaux de pointeurs, des tableaux alloués sur le tas et d'un certain nombre d'autres collections. Vous découvrirez quelques structures de données complexes, mais vous en apprendrez plus en lisant *C++ Unleashed*, publié par Sams Publishing. Vous pouvez également consulter l'annexe E de cet ouvrage.

Deux aspects importants de la programmation sont qu'il y a toujours plus de choses à apprendre et qu'il existe toujours des livres permettant de les apprendre.

Construire des tableaux de pointeurs

Jusqu'à présent, les tableaux utilisés stockaient leurs éléments dans la pile et, comme vous le savez, l'espace disponible dans la pile est limité. Pour plus de liberté, vous pouvez déclarer les objets sur le tas et ne stocker dans le tableau que des pointeurs vers ces objets. La quantité requise d'espace mémoire s'en trouvera ainsi considérablement réduite. Le Listing 13.6 effectue les mêmes opérations que le programme du Listing 13.4, mais stocke tous les objets sur le tas. Le tableau peut alors passer de 5 à 500 éléments. Il s'appelle désormais `Famille` et non plus `Portee`.

Listing 13.6 : Stockage d'un tableau dans le tas

```
0:  // Listing 13.6 - Un tableau de pointeurs sur les objets
1:
2:  #include <iostream>
3:  using namespace std;
4:
```

```
 5:  class Chat
 6:  {
 7:  public:
 8:      Chat() { sonAge = 1; sonPoids = 5; }
 9:      ~Chat() {}                              // destructeur
10:     int GetAge() const { return sonAge; }
11:     int GetPoids() const { return sonPoids; }
12:     void SetAge(int age) { sonAge = age; }
13:
14:  private:
15:     int sonAge;
16:     int sonPoids;
17:  };
18:
19:  int main()
20:  {
21:     Chat * Famille[500];
22:     int i;
23:     Chat * pChat;
24:     for (i = 0; i < 500; i++)
25:     {
26:        pChat = new Chat;
27:        pChat->SetAge(2*i +1);
28:        Famille[i] = pChat;
29:     }
30:
31:     for (i = 0; i < 500; i++)
32:     {
33:        cout << "Chat numéro " << i+1 << ": ";
34:        cout << Famille[i]->GetAge() << endl;
35:     }
36:     return 0;
37:  }
```

Ce programme produit le résultat suivant :

```
Chat numéro 1 : 1
Chat numéro 2 : 3
Chat numéro 3 : 5
...
Chat numéro 499 : 997
Chat numéro 500 : 999
```

L'objet `Chat` déclaré de la ligne 5 à la ligne 17 est identique à celui du Listing 13.4. Cette fois-ci, cependant, le tableau s'appelle `Famille` et comprend 500 éléments, qui sont des pointeurs vers des objets de la classe `Chat`.

Dans la boucle initiale (de la ligne 24 à la ligne 29), les 500 objets `Chat` sont créés sur le tas. L'âge de chacun est égal à deux fois la valeur d'indice, plus un : l'âge du premier objet `Chat` est égal à 1, le deuxième à 3, le troisième à 5, etc. Après la création du pointeur, la ligne 28 affecte le pointeur au tableau. Le tableau ayant été déclaré comme un tableau de pointeurs, c'est le pointeur et non la valeur déréférencée de ce dernier qui est ajouté au tableau.

La seconde boucle permet d'afficher les valeurs du tableau (lignes 31 à 35). La ligne 33 affiche un nombre pour indiquer l'objet concerné. Les décalages d'indice partant de zéro, la ligne 33 ajoute 1 pour un comptage commençant à 1. À la ligne 34, on accède au pointeur à l'aide de l'indice `i` du tableau `Famille`. Cette adresse permet alors d'appeler la méthode `GetAge()` de l'objet concerné.

Dans cet exemple, le tableau `Famille` et ses pointeurs sont stockés dans la pile, alors que les 500 objets `Chat` sont créés sur le tas.

L'arithmétique des pointeurs, un sujet complexe

Les pointeurs ont été introduits au Chapitre 8. Avant de poursuivre avec les tableaux, revenons un instant sur ce sujet pour comprendre l'arithmétique des pointeurs.

Les pointeurs ne permettent que peu de choses du point de vue mathématique ; ils peuvent être soustraits les uns des autres. Une technique puissante consiste à faire pointer deux pointeurs vers des éléments différents d'un tableau et à calculer la différence entre ces deux pointeurs pour savoir combien d'éléments se trouvent entre les deux. Cette technique est particulièrement utile pour analyser des tableaux de caractères, comme le montre le Listing 13.7.

Listing 13.7 : Analyse des mots à partir d'une chaîne de caractères

```
 0:  #include <iostream>
 1:  #include <ctype.h>
 2:  #include <string.h>
 3:
 4:  bool LireMot(char* chaine,
 5:                char* mot, int& motDecale);
 6:
 7:  // Programme principal
 8:  int main()
 9:  {
10:     const int tailleTampon = 255;
11:     char tampon[tailleTampon+1];   // contient la chaîne
12:     char mot[tailleTampon+1];      // contient le mot
```

```
13:     int motDecale = 0;              // commencer au début
14:
15:     std::cout << "Entrez une chaîne : ";
16:     std::cin.getline(tampon, tailleTampon);
17:
18:     while (LireMot(tampon, mot, motDecale))
19:     {
20:        std::cout << "Mot obtenu : " << mot << std::endl;
21:     }
22:     return 0;
23:  }
24:
25:  // Fonction pour analyser les mots d'une chaîne.
26:  bool LireMot(char* chaine, char* mot, int& motDecale)
27:  {
28:   if (chaine[motDecale] == 0)  // fin de la chaîne ?
29:     return false;
30:
31:   char *p1, *p2;
32:   p1 = p2 = chaine + motDecale; // pointe sur le mot suivant
33:
34:   // effacer les premiers espaces
35:   for (int i = 0; i < (int)strlen(p1) && !isalnum(p1[0]);
35a:       i++)
36:     p1++;
37:
38:   // voir s'il y a un mot
39:   if (!isalnum(p1[0]))
40:      return false;
41:
42:   // p1 pointe maintenant pour commencer le mot suivant
43:   // point p2 ici aussi
44:   p2 = p1;
45:
46:   // amener p2 à la fin du mot
47:   while (isalnum(p2[0]))
48:      p2++;
49:
50:   // p2 est maintenant à la fin du mot
51:   // p1 est au début du mot
52:   // la longueur du mot = la différence
53:   int lg = int (p2 - p1);
54:
55:   // copier le mot dans le tampon
56:   strncpy (mot, p1, lg);
57:
58:   // null termine la chaîne
```

```
59:   mot[lg] = '\0';
60:
61:     // trouver le début du mot suivant
62:     for (int j = int(p2 - chaine); j < (int)strlen(chaine)
63:                                    && !isalnum(p2[0]); j++)
64:     {
65:        p2++;
66:     }
67:
68:     motDecale = int(p2 - chaine);
69:
70:     return true;
71:  }
```

Ce programme produit le résultat suivant :

```
Entrez une chaîne : ce code est apparu pour la premiere fois dans C++ Report
Mot obtenu : ce
Mot obtenu : code
Mot obtenu : est
Mot obtenu : apparu
Mot obtenu : pour
Mot obtenu : la
Mot obtenu : premiere
Mot obtenu : fois
Mot obtenu : dans
Mot obtenu : C
Mot obtenu : Report
```

Ce programme permet de saisir une phrase, qui est ensuite décomposée en mots (ensembles de caractères alphanumériques). La ligne 15 demande à l'utilisateur d'entrer une chaîne, qui est ensuite passée à la méthode `LireMot()` ligne 18 avec un tampon permettant de conserver le premier mot et une variable entière appelée `motDecale`, initialisée à zéro à la ligne 13.

`LireMot()` renvoie chaque mot de la chaîne. Ces mots sont affichés la ligne 20 jusqu'à ce que `LireMot()` renvoie `false`.

Chaque appel à `LireMot()` entraîne un saut à la ligne 26. La ligne 28 teste si la valeur de `chaine[motDecale]` est égale à zéro. Ce sera vrai si vous êtes à la fin de la chaîne, auquel cas `LireMot()` renvoie `false`. `cin.getline()` vérifie que la chaîne entrée se termine par un caractère nul, c'est-à-dire le caractère `\0`.

La ligne 31 déclare deux pointeurs de caractères, `p1` et `p2`. Ils sont initialiés à la ligne 32 afin de pointer à l'indice `motDecale` de la chaîne. Initialement, `motDecale` valant zéro, ces pointeurs pointent donc au début de la chaîne.

Les lignes 35 et 36 parcourent la chaîne, en poussant `p1` jusqu'au premier caractère alphanumérique. Les lignes 39 et 40 s'assurent que l'on en trouve un, sinon `false` est renvoyé.

`p1` pointe maintenant sur le début du mot suivant et la ligne 44 initialise `p2` de sorte qu'il pointe vers cette même position.

Les lignes 47 et 48 font parcourir le mot par `p2` afin qu'il s'arrête au premier caractère qui n'est pas alphanumérique. `p2` pointe donc maintenant à la fin du mot dont le début est pointé par `p1`. En soustrayant `p1` de `p2` (ligne 53) et en transtypant le résultat en un entier, on peut donc connaître la longueur du mot, qui est ensuite copiée dans le tampon `mot` à l'aide d'une fonction de copie de chaîne fournie par la bibliothèque standard.

La ligne 59 ajoute un caractère nul pour marquer la fin du mot. Puis, `p2`est incrémenté pour pointer au début du mot suivant et le décalage de ce mot est poussé dans la référence entière `motDecale`. Enfin, `true` est renvoyé pour indiquer qu'on a trouvé un mot.

Il s'agit d'un exemple classique de code qui est plus compréhensible si l'on utilise un débogueur afin de suivre son exécution pas à pas.

L'arithmétique des pointeurs est employée à divers endroits de ce listing. En soustrayant un pointeur d'un autre (comme à la ligne 53), on détermine le nombre d'éléments qui les sépare. En outre, la ligne 55 montre que l'incrémentation d'un pointeur le fait passer à l'élément suivant du tableau au lieu de simplement lui ajouter un. L'arithmétique des pointeurs est très utilisée avec les pointeurs et les tableaux, mais elle représente aussi une activité dangereuse et doit donc être considérée avec circonspection.

Déclaration de tableaux sur le tas

Il est possible de stocker un tableau entier sur le tas. Pour cela, vous devez créer un pointeur vers le tableau en utilisant le mot-clé `new` et l'opérateur d'indexation. Vous obtenez ainsi un pointeur vers une zone du tas contenant le tableau. La ligne suivante, par exemple :

```
Chat *Famille = new Chat[500];
```

déclare `Famille` comme un pointeur sur le premier élément d'un tableau de 500 objets `Chat`. En d'autres termes, `Famille` pointe sur – ou contient l'adresse de – `Famille[0]`.

L'avantage de déclarer `Famille` de cette façon est que cela vous permet d'utiliser l'arithmétique des pointeurs pour accéder à chaque élément de `Famille`. Vous pouvez par exemple écrire :

```
Chat *Famille = new Chat[500];
Chat *pChat = Famille;            // pChat pointe sur Famille[0]
pChat->SetAge(10);                // définit Famille[0] à 10
pChat++;                          // pointe sur Famille[1]
pChat->SetAge(20);                // définit Famille[1] à 20
```

Ces lignes déclarent un tableau de 500 objets `Chat` et un pointeur vers le premier élément de ce tableau. En utilisant ce pointeur, on peut appeler la méthode `SetAge()` en lui passant la valeur 10. Puis, ce pointeur est incrémenté pour désigner l'objet `Chat` suivant du tableau. On appelle ensuite à nouveau la méthode `SetAge()` en lui passant cette fois-ci la valeur 20.

Pointeurs sur tableau et tableaux de pointeurs

Examinez les trois déclarations suivantes :

```
1:  Chat   Famille1[500];
2:  Chat * Famille2[500];
3:  Chat * Famille3 = new Chat[500];
```

`Famille1` est un tableau de 500 objets `Chat`, alors que `Famille2` est un tableau de 500 pointeurs sur des objets de la classe `Chat`. Enfin, `Famille3` est un pointeur sur un tableau contenant 500 objets `Chat`.

Les différences entre ces trois lignes de code affectent énormément le comportement de ces tableaux. Ce qui peut sembler le plus surprenant est que `Famille3` est une variante de `Famille1`, alors qu'il est très différent de `Famille2`.

Ceci soulève le problème épineux de la relation entre les pointeurs et les tableaux. Dans le troisième exemple, `Famille3` est un pointeur sur un tableau. L'adresse contenue dans ce pointeur correspond à l'adresse du premier élément du tableau et c'est exactement la même chose pour `Famille1`.

Pointeurs et noms de tableaux

En C++, un nom de tableau est un pointeur constant sur le premier élément du tableau. Par conséquent, dans la déclaration suivante :

```
Chat Famille[50];
```

`Famille` est un pointeur sur `&Famille[0]`, qui est l'adresse du premier élément du tableau `Famille`.

Vous avez le droit d'utiliser des noms de tableaux comme des pointeurs constants et réciproquement. `Famille + 4`, par exemple, est une expression permettant d'accéder à l'élément `Famille[4]`.

Le compilateur se charge d'effectuer les calculs lorsque vous ajoutez, incrémentez et décrémentez des pointeurs. Lorsque vous écrivez `Famille + 4`, l'adresse obtenue ne se

situe pas 4 octets plus loin, mais 4 objets plus loin. Si chaque objet occupe 4 octets, `Famille + 4` indiquera donc un déplacement de 16 octets par rapport au début du tableau. Si chaque objet `Chat` contient 4 variables membres de type `long` et 2 variables membres de type `short`, la taille d'un chat sera donc de 20 octets ((4×4)+(2×2)). En ce cas, `Famille + 4` correspondra à un déplacement de 80 octets après le début du tableau.

Le Listing 13.8 montre comment déclarer et utiliser un tableau stocké sur le tas.

Listing 13.8 : Création d'un tableau sur le tas à l'aide de *new*

```
 0:  // Listing 13.8 - Création d'un tableau sur le tas
 1:
 2:  #include <iostream>
 3:
 4:  class Chat
 5:  {
 6:  public:
 7:      Chat() { sonAge = 1; sonPoids = 5; }
 8:      ~Chat();
 9:     int GetAge() const { return sonAge; }
10:     int GetPoids() const { return sonPoids; }
11:     void SetAge(int age) { sonAge = age; }
12:
13:  private:
14:     int sonAge;
15:     int sonPoids;
16:  };
17:
18:  Chat :: ~Chat()
19:  {
20:     // std::cout << "Appel du destructeur !\n";
21:  }
22:
23:  int main()
24:  {
25:     Chat * Famille = new Chat[500];
26:     int i;
27:
28:     for (i = 0; i < 500; i++)
29:     {
30:        Famille[i].SetAge(2*i +1);
31:     }
32:
33:     for (i = 0; i < 500; i++)
34:     {
35:        std::cout << "Chat numéro " << i+1 << " : ";
```

```
36:        std::cout << Famille[i].GetAge() << std::endl;
37:     }
38:
39:     delete [] Famille;
40:
41:     return 0;
42:  }
```

Ce programme produit le résultat suivant :

```
Chat numéro 1 : 1
Chat numéro 2 : 3
Chat numéro 3 : 5
...
Chat numéro 499 : 997
Chat numéro 500 : 999
```

La ligne 25 déclare `Famille`, qui est un pointeur vers un tableau de 500 objets `Chat`. Le tableau complet est créé sur le tas par un appel à `new Chat[500]`.

En ligne 30, vous pouvez constater que le pointeur que vous avez déclaré peut être utilisé avec l'opérateur d'indexation `[]` et qu'il peut donc être manipulé comme un tableau ordinaire. La ligne 36 l'utilise à nouveau pour appeler la méthode `GetAge()`. D'un point de vue pratique, vous pouvez considérer ce pointeur vers le tableau `Famille` comme un nom de tableau, mais il ne faut pas oublier de libérer la mémoire que vous aviez allouée lors de la configuration du tableau. Cette opération est réalisée à la ligne 39 par un appel à `delete`.

Suppression des tableaux stockés sur le tas

Que devient la mémoire allouée aux objets `Chat` lorsque le tableau est détruit ? Il y a-t-il un risque de fuite mémoire ?

La suppression de `Famille` libère automatiquement toute la mémoire allouée au tableau si vous utilisez `delete` avec l'opérateur `[]`. Lorsqu'il voit les crochets, le compilateur est suffisamment intelligent pour savoir qu'il doit détruire chaque objet du tableau et restituer sa mémoire au tas.

Pour le constater, modifiez la taille du tableau pour la faire passer de 500 à 10 éléments (lignes 27, 28 et 33). Puis supprimez les barres obliques de commentaire devant l'instruction `cout`, à la ligne 20. Lorsque le programme atteint la ligne 39 et détruit le tableau, vous verrez apparaitre les différents appels aux destructeurs des objets `Chat`.

Lorsqu'un élément est créé sur le tas à l'aide du mot-clé `new`, il doit toujours être supprimé et sa mémoire libérée avec mot-clé `delete`. De même, si vous créez un tableau à l'aide de `new <classe>[taille]`, vous devez supprimer le tableau et libérer sa mémoire à l'aide de `delete[]`. Les crochets indiquent au compilateur que l'objet à supprimer est un tableau.

Si vous omettez les crochets, seul le premier objet du tableau sera supprimé. Amusez-vous à supprimer les crochets de l'instruction `delete` figurant à la ligne 39. Si vous avez modifié la ligne 20 pour que l'appel du destructeur soit affiché, vous pourrez constater qu'un seul objet `Chat` est supprimé. Félicitations ! Vous venez de produire une fuite mémoire.

Redimensionner les tableaux en cours d'exécution

Le plus gros avantage d'allouer des tableaux sur le tas est qu'il est possible de déterminer la taille du tableau au moment de l'exécution, avant de l'allouer. Si, par exemple, vous avez demandé à l'utilisateur d'entrer la taille d'une famille dans une variable appelée `TailleFamille`, vous pourrez ensuite déclarer un tableau `Chat` de la façon suivante :

```
Chat *pFamille = new Chat[TailleFamille];
```

Vous avez maintenant un pointeur vers un tableau d'objets `Chat`. Vous pouvez ensuite créer un pointeur vers le premier élément et parcourir ce tableau à l'aide d'un pointeur et de l'arithmétique des pointeurs :

```
Chat *pChatActuel = Famille[0];
for ( int i = 0; i < TailleFamille; i++, pChatActuel++ )
{
   pChatActuel->SetAge(i);
};
```

C++ considérant les tableaux comme des pointeurs spéciaux, vous pouvez ignorer le deuxième pointeur et utiliser simplement l'indexation classique des tableaux :

```
for (int i = 0; i < TailleFamille; i++)
{
   pFamille[i].SetAge(i);
};
```

L'utilisation des crochets déréférence automatiquement le pointeur et le compilateur réalise le calcul approprié en utilisant l'arithmétique des pointeurs. Vous pouvez également utiliser une technique semblable pour redimensionnement le tableau en cours d'exécution si vous manquez de place, comme dans le Listing 13.9.

Listing 13.9 : Réallocation d'un tableau en cours d'exécution

```
 0:  //Listing 13.9
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:  int main()
 5:  {
 6:    int  TailleAllocation = 5;
 7:    int *pTableauDeNombres = new int[TailleAllocation];
 8:    int  ElementsUtilises = 0;
 9:    int  ElementsMaximumAutorises = TailleAllocation;
10:    int  NombreSaisi = -1;
11:
12:    cout << endl << "Nombre suivant = ";
13:    cin >> NombreSaisi;
14:
15:    while ( NombreSaisi > 0 )
16:    {
17:       pTableauDeNombres[ElementsUtilises++] = NombreSaisi;
18:
19:       if ( ElementsUtilises == ElementsMaximumAutorises )
20:       {
21:        int *pGrandTableau =
22:           new int[ElementsMaximumAutorises+TailleAllocation];
23:
24:        for ( int IndiceCopie = 0;
25:              IndiceCopie < ElementsMaximumAutorises;
26:              IndiceCopie ++ )
27:        {
28:            pGrandTableau[IndiceCopie] =
28a             pTableauDeNombres[IndiceCopie];
29:        }
30:
31:        delete [] pTableauDeNombres;
32:        pTableauDeNombres = pGrandTableau;
33:        ElementsMaximumAutorises+= TailleAllocation;
34:      }
35:      cout << endl << "Nombre suivant = ";
36:      cin >> NombreSaisi;
37:    }
38:
39:    for (int Index = 0; Index < ElementsUtilises; Index++)
40:    {
41:      cout << pTableauDeNombres[Index] << endl;
42:    }
43:    return 0;
44:  }
```

Ce programme produit le résultat suivant :

```
Nombre suivant = 10

Nombre suivant = 20

Nombre suivant = 30

Nombre suivant = 40

Nombre suivant = 50

Nombre suivant = 60

Nombre suivant = 70

Nombre suivant = 0
10
20
30
40
50
60
70
```

Dans cet exemple, les nombres sont saisis les uns après les autres et stockés dans un tableau. Lorsqu'un nombre saisi est inférieur ou égal à 0, le tableau qui a été constitué est affiché.

Les lignes 6 à 9 déclarent plusieurs variables. Plus précisément, la taille initiale du tableau est fixée à 5 à la ligne 6, puis le tableau est alloué à la ligne 7 et son adresse est affectée à `pTableauDeNombres`.

Les lignes 12 et 13 récupèrent le premier nombre entré et le placent dans la variable `NombreSaisi`. À la ligne 15, si le nombre entré est supérieur à zéro, le traitement a lieu. Dans le cas contraire, le programme passe à la ligne 38.

La ligne 17 place `NombreSaisi` dans le tableau. Ceci ne posera pas de problème la première fois car vous savez qu'il y a de la place. La ligne 19 teste si c'est le dernier élément pour lequel le tableau a de la place. S'il reste de la place, le contrôle passe à la ligne 35 ; sinon, le corps de l'instruction `if` est exécuté pour augmenter la taille du tableau (lignes 20-34).

Un nouveau tableau est créé à la ligne 21 pour contenir cinq éléments (`TailleAllocation`) de plus que le tableau courant. Les lignes 24 à 29 copient ensuite l'ancien tableau dans le nouveau, à l'aide de la notation des tableaux (vous pourriez utiliser l'arithmétique des pointeurs).

La ligne 31 supprime l'ancien tableau et la ligne 32 remplace l'ancien pointeur par celui du nouveau tableau. La ligne 33 augmente `ElementsMaximumAutorises` pour l'adapter à la nouvelle taille.

Les lignes 39 à 42 affichent le résultat.

Faire	*Ne pas faire*
● Se rappeler qu'un tableau de *n* éléments est indicé de 0 à `n-1`. ● Utiliser l'indexation des tableaux avec les pointeurs qui pointent sur des tableaux. ● Utiliser `delete[]` pour supprimer un tableau créé sur le tas. `delete` (sans les crochets) ne supprime que le premier élément.	● Écrire ou lire après la fin d'un tableau. ● Confondre un tableau de pointeurs et un pointeur sur un tableau. ● Oublier de supprimer la mémoire allouée avec `new`.

Tableaux de caractères et chaînes

Une chaîne "de type C" est un tableau de caractères qui se termine par un caractère nul. Jusqu'à présent, les seules chaînes de ce type que vous avez rencontrées étaient les constantes chaînes anonymes utilisées dans les instructions d'affichage comme celle-ci :

```
cout << "bonjour";
```

Vous pouvez déclarer et initialiser une chaîne de type C comme vous le feriez pour n'importe quel tableau. Voici un exemple :

```
char Salut[] = { 'b', 'o', 'n', 'j', 'o', 'u' ,'r','\0' };
```

Ici, `Salut` est déclaré comme un tableau de caractères et initialisé avec plusieurs caractères. Le dernier caractère, `'\0'`, est le caractère nul, qui est le caractère de terminaison des chaînes de type C. Bien que cette approche caractère par caractère fonctionne, elle est difficile à saisir et est bien trop sujette aux erreurs de frappe. C permet d'utiliser la forme suivante, qui est un équivalent raccourci de la précédente :

```
char Salut[] = "Bonjour";
```

Il faut noter deux choses dans cette syntaxe :

- Les caractères entre apostrophes simples, séparés par des virgules, ont été remplacés par une chaîne de caractères entre guillemets, sans accolades.
- Il n'est pas nécessaire d'ajouter le caractère nul à la fin car le compilateur s'en charge pour vous.

Lorsque vous déclarez une chaîne, vous devez vérifier qu'elle pourra contenir ce que vous souhaitez y mettre. La longueur d'une chaîne de type C est le nombre de ses caractères, y compris le caractère nul. La chaîne *Bonjour* occupe donc 8 octets : le mot occupe 7 octets et le caractère nul 1 octet.

Vous pouvez aussi créer des tableaux de caractères non initialisés. Comme pour tout autre tableau, vous devez vérifier que vous n'en mettez pas plus qu'il y a d'éléments réservés. Le Listing 13.10 met en œuvre un tampon non initialisé.

Listing 13.10 : Remplissage d'un tableau

```
0:  //Listing 13.10 Tampons de caractères
1:
2:  #include <iostream>
3:
4:  int main()
5:  {
6:      char tampon[80];
7:      std::cout << "Entrez la chaîne : ";
8:      std::cin >> tampon;
9:      std::cout << "Contenu : " << tampon << std::endl;
10:     return 0;
11:  }
```

Ce programme produit le résultat suivant :

```
Entrez la chaîne : Bonjour les amis
Contenu : Bonjour
```

La ligne 6 indique que le tampon peut recevoir 80 caractères, soit 79 caractères et 1 caractère nul.

La ligne suivante invite l'utilisateur à entrer une chaîne qui est stockée dans le tampon (ligne 8). `cin` place un caractère nul immédiatement après la fin du mot transféré dans le tampon.

Ce programme contient deux erreurs potentielles. La première est que si l'utilisateur tape plus de 79 caractères, `cin` écrira après la fin du tampon. La seconde est qu'un caractère d'espacement est considéré par `cin` comme une fin de chaîne et qu'il arrête alors d'écrire dans le tampon.

Pour résoudre ces problèmes, vous devez appeler la méthode spéciale `get()` sur `cin`, en lui passant les trois paramètres suivants :

- le tampon à remplir ;
- le nombre maximal de caractères à lire ;
- le délimiteur de fin de saisie.

Par défaut, le délimiteur est le caractère nouvelle ligne, comme le montre le Listing 13.11.

Listing 13.11 : Remplir un tableau avec un nombre maximal de caractères

```
0:   //Listing 13.11 - Utiliser cin.get()
1:
2:   #include <iostream>
3:   using namespace std;
4:
5:   int main()
6:   {
7:      char tampon[80];
8:      cout << "Entrez une chaîne : ";
9:      cin.get(tampon, 79); // Lit 79 car max ou nouvelle ligne
10:     cout << "Contenu du tampon : " << tampon << endl;
11:     return 0;
12:  }
```

Ce programme produit le résultat suivant :

```
Entrez une chaîne : Bonjour les amis
Contenu du tampon : Bonjour les amis
```

Le tampon déclaré à la ligne 7 est passé comme premier paramère à la méthode `get()` de `cin` (ligne 9). Le second paramètre correspond au nombre maximum de caractères à lire. Ici, il ne doit pas dépasser 79 caractères pour laisser une place au caractère nul. Il est inutile de fournir un caractère de terminaison car, ici, la valeur par défaut (nouvelle ligne) suffit.

Si vous entrez des espaces, des tabulations ou d'autres sortes d'espaces, ils seront affectés à la chaîne. La saisie se terminera si vous entrez un caractère de nouvelle ligne ou que vous avez saisi 79 caractères. Vous pouvez le vérifier en relançant ce programme et en essayant d'entrer une chaîne supérieure à 79 caractères.

Les fonctions *strcpy()* et *strncpy()*

La bibliothèque C++ fournit plusieurs fonctions permettant de traiter des chaînes. C++ hérite de son ancêtre C les fonctions de manipulations des chaînes de type C. Parmi elles, `strcpy()` et `strncpy()` permettent de copier une chaîne vers une autre. `strcpy()` copie le contenu entier d'une chaîne dans un tampon, tandis que `strncpy()` ne copie qu'un nombre donné de caractères. Le Listing 13.12 montre comment utiliser `strcpy()`.

Listing 13.12 : Utilisation de la fonction *strcpy()*

```
 0:  //Listing 13.12 Utiliser strcpy()
 1:
 2:  #include <iostream>
 3:  #include <string.h>
 4:  using namespace std;
 5:
 6:  int main()
 7:  {
 8:     char Chaine1[] = "Nul n'est prophete en son pays";
 9:     char Chaine2[80];
10:
11:     strcpy(Chaine2, Chaine1);
12:
13:     cout << "Chaine1 : " << Chaine1 << endl;
14:     cout << "Chaine2 : " << Chaine2 << endl;
15:     return 0;
16:  }
```

Ce programme produit le résultat suivant :

```
Chaine1 : Nul n'est prophete en son pays
Chaine2 : Nul n'est prophete en son pays
```

Ce programme est relativement simple. Il copie les données d'une chaîne dans une autre. Le fichier en-tête `string.h` inclus à la ligne 3 contient le prototype de la fonction `strcpy()`. Cette dernière prend en paramètre deux tableaux de caractères, le tableau cible suivi du tableau source de la copie. La ligne 11 appelle cette fonction pour `Chaine1` en `Chaine2`.

Le fonction `strcpy()` présente un inconvénient majeur puisqu'elle peut copier les caractères au-delà du dernier élément du tableau cible si le tableau source est plus grand. `strncpy()` permet de remédier à ce problème puisqu'elle permet de limiter le nombre des caractères copiés. Elle effectue donc la copie dans le tampon de destination jusqu'à rencontrer un caractère nul ou atteindre le nombre maximal de caractères dans le tampon source. Le Listing 13.13 montre comment utiliser cette fonction.

Listing 13.13 : Utilisation de la fonction *strncpy()*

```
0:  //Listing 13.13 Utiliser strncpy()
1:
2:  #include <iostream>
3:  #include <string.h>
4:
```

```
 5:  int main()
 6:  {
 7:     const int LongMax = 80;
 8:     char Chaine1[] = "Nul n'est prophete en son pays";
 9:     char Chaine2[LongMax+1];
10:
11:     strncpy(Chaine2, Chaine1, LongMax);
12:
13:     std::cout << "Chaine1 : " << Chaine1 << std::endl;
14:     std::cout << "Chaine2 : " << Chaine2 << std::endl;
15:     return 0;
16:  }
```

Ce programme produit le résultat suivant :

```
Chaine1 : Nul n'est prophete en son pays
Chaine2 : Nul n'est prophete en son pays
```

Comme le précédent, ce programme copie le contenu d'une chaîne dans une autre. Cependant, la ligne 12 a été modifiée puisqu'elle appelle désormais la fonction `strncpy()`. Celle-ci utilise un troisième paramètre, qui est le nombre maximal de caractères à copier. La taille du tampon `Chaine2` est égale à `LongMax+1` caractères. Le caractère supplémentaire correspond au caractère nul qui est ajouté automatiquement à la fin de la chaîne.



Comme avec le tableau d'entiers du Listing 13.9, les tableaux de caractères peuvent être redimensionnés à l'aide des techniques d'allocation sur le tas et de la copie, élément par élément. Les classes de chaînes de C++, plus souples, utilisent une variante de cette technique pour permettre aux chaînes de grandir et de se réduire en fonction des besoins, ou pour insérer ou supprimer des éléments au milieu de la chaîne.

La classe *String*

Héritées du langage C, les chaînes terminées par un caractère nul et la bibliothèque de fonctions (parmi lesquelles figure `strcpy()`) s'intègrent mal à une structure de programme orientée objet. La bibliothèque standard de C++ inclut une classe `String` qui fournit un ensemble encapsulé de données et de méthodes permettant de masquer les détails internes aux clients de la classe.

Avant d'utiliser cette classe, nous allons créer une classe `String` personnalisée qui nous permettra de comprendre les problèmes à résoudre. Notre classe `String` doit au mininum surmonter les limites des tableau de caractères.

Comme tous les tableaux, un tableau de caractères est statique. Le programmeur doit donc définir sa taille en mémoire, même s'il ne l'utilise pas en totalité. Comme nous l'avons vu, toute tentative d'écriture au-delà du dernier élément est désastreuse.

Une classe `String` bien conçue ne doit utiliser que l'espace mémoire nécessaire et suffisant. Si elle n'arrive pas à allouer de la mémoire, l'application doit se terminer proprement.

Le Listing 13.14 présente une première approche de notre classe `String`.



Cette classe `String` personnalisée a ses limites et ne doit pas être utilisée en l'état dans une application commerciale. La bibliothèque standard en propose une qui offre toutes les garanties.

Listing 13.14 : Création d'une classe *String*

```
 0:  //Listing 13.14 Utilisation d'une classe String
 1:
 2:  #include <iostream>
 3:  #include <string.h>
 4:  using namespace std;
 5:
 6:  // Classe String rudimentaire
 7:  class String
 8:  {
 9:  public:
10:     // constructeurs
11:     String();
12:     String(const char *const);
13:     String(const String &);
14:     ~String();
15:
16:     // opérateurs surchargés
17:     char & operator[](unsigned short indice);
18:     char operator[](unsigned short indice) const;
19:     String operator+(const String&);
20:     void operator+=(const String&);
21:     String & operator= (const String &);
22:
23:     // méthodes d'accès
24:     unsigned short GetLen() const { return saLongueur; }
25:     const char * GetString() const { return saChaine; }
26:
27:  private:
28:     String (unsigned short);         // constructeur privé
29:     char * saChaine;
```

```
30:     unsigned short saLongueur;
31:  };
32:
33:  // le constructeur par défaut crée une chaîne de 0 octet
34:  String::String()
35:  {
36:     saChaine = new char[1];
37:     saChaine[0] = '\0';
38:     saLongueur = 0;
39:  }
40:
41:  // constructeur (utilitaire) privé, utilisé seulement par
42:  // les méthodes de la classe pour créer une nouvelle chaîne
43:  // de la taille requise. Complétée par des zéros.
44:  String::String(unsigned short lg)
45:  {
46:     saChaine = new char[lg+1];
47:     for (unsigned short i = 0; i <= lg; i++)
48:        saChaine[i] = '\0';
49:     saLongueur = lg;
50:  }
51:
52:  // Convertit un tableau de caractères en une chaîne
53:  String::String(const char * const chaineC)
54:  {
55:     saLongueur = strlen(chaineC);
56:     saChaine = new char[saLongueur+1];
57:     for (unsigned short i = 0; i < saLongueur; i++)
58:        saChaine[i] = chaineC[i];
59:     saChaine[saLongueur]='\0';
60:  }
61:
62:  // constructeur de copie
63:  String::String (const String & rhs)
64:  {
65:     saLongueur = rhs.GetLen();
66:     saChaine = new char[saLongueur+1];
67:     for (unsigned short i = 0; i < saLongueur; i++)
68:        saChaine[i] = rhs[i];
69:     saChaine[saLongueur] = '\0';
70:  }
71:
72:  // destructeur, libère toute la mémoire allouée
73:  String::~String ()
74:  {
75:     delete [] saChaine;
76:     saLongueur = 0;
```

```
 77:  }
 78:
 79:  // opérateur d'affectation, libère la mémoire existante
 80:  // puis copie la chaîne et la taille
 81:  String& String::operator=(const String & rhs)
 82:  {
 83:     if (this == &rhs)
 84:        return *this;
 85:     delete [] saChaine;
 86:     saLongueur = rhs.GetLen();
 87:     saChaine = new char[saLongueur+1];
 88:     for (unsigned short i = 0; i < saLongueur; i++)
 89:        saChaine[i] = rhs[i];
 90:     saChaine[saLongueur] = '\0';
 91:     return *this;
 92:  }
 93:
 94:  // opérateur d'indexation non constant, renvoie une
 95:  // référence au caractère pour qu'il puisse
 96:  // être modifié !
 97:  char & String::operator[](unsigned short indice)
 98:  {
 99:     if (indice > saLongueur)
100:        return saChaine[saLongueur-1];
101:    else
102:        return saChaine[indice];
103:  }
104:
105:  // opérateur d'indexation constant pour utilisation sur
106:  // les objets constants (voir le constructeur de copie !)
107:  char String::operator[](unsigned short indice) const
108:  {
109:    if (indice > saLongueur)
110:        return saChaine[saLongueur-1];
111:    else
112:        return saChaine[indice];
113:  }
114:
115:  // crée une nouvelle chaîne en ajoutant
116:  // la chaîne actuelle à rhs
117:  String String::operator+(const String& rhs)
118:  {
119:    unsigned short lgTotale = saLongueur + rhs.GetLen();
120:    String temp(lgTotale);
121:    unsigned short i;
122:    for ( i= 0; i < saLongueur; i++)
123:        temp[i] = saChaine[i];
```

```
124:    for (unsigned short j = 0; j < rhs.GetLen(); j++, i++)
125:        temp[i] = rhs[j];
126:    temp[lgTotale]='\0';
127:    return temp;
128:  }
129:
130:  // modifie la chaîne actuelle, ne renvoie rien
131:  void String::operator+=(const String& rhs)
132:  {
133:    unsigned short lgRhs = rhs.GetLen();
134:    unsigned short lgTotale = saLongueur+ lgRhs;
135:    String  temp(lgTotale);
136:    unsigned short i;
137:    for (i = 0; i < saLongueur; i++)
138:        temp[i] = saChaine[i];
139:    for (unsigned short j = 0; j < lgRhs; j++, i++)
140:        temp[i] = rhs[i-saLongueur];
141:    temp[lgTotale]='\0';
142:    *this = temp;
143:  }
144:
145:  int main()
146:  {
147:    String s1("Test initial");
148:    cout << "S1 :\t" << s1.GetString() << endl;
149:
150:    char * temp = "Bonjour tout le monde ";
151:    s1 = temp;
152:    cout << "S1 :\t" << s1.GetString() << endl;
153:
154:    char temp2[20];
155:    strcpy(temp2,"; comment ca va ?");
156:    s1 += temp2;
157:    cout << "temp2 :\t" << temp2 << endl;
158:    cout << "S1 :\t" << s1.GetString() << endl;
159:
160:    cout << "S1[4] :\t" << s1[4] << endl;
161:    s1[4]= 'x';
162:    cout << "S1 :\t" << s1.GetString() << endl;
163:
164:    cout << "S1[999] :\t" << s1[999] << endl;
165:
166:    String s2(" Autre chaîne");
167:    String s3;
168:    s3 = s1 + s2;
169:    cout << "S3 :\t" << s3.GetString() << endl;
170:
```

```
171:    String s4;
172:    s4 = "Pourquoi ca marche ?";
173:    cout << "S4 :\t" << s4.GetString() << endl;
174:    return 0;
175:  }
```

Ce programme produit le résultat suivant :

```
S1 :        Test initial
S1 :        Bonjour tout le monde
temp2 :     ; comment ca va ?
S1 :        Bonjour tout le monde ; comment ca va ?
S1[4] :     o
S1 :        Bonjxur tout le monde ; comment ca va ?
S1[999] :   ?
S3 :        Bonjxur tout le monde ; comment ca va ? Autre chaîne
S4 :        Pourquoi ca marche ?
```

La déclaration de notre classe `String` est comprise de la ligne 7 à la ligne 31. Pour ajouter de la souplesse à la classe, il y a trois constructeurs aux lignes 11 à 13 : le constructeur par défaut, le constructeur de copie et un constructeur prenant en paramètre une chaîne existante terminée par un caractère nul (style C).

Pour permettre aux utilisateurs de manipuler facilement les chaînes, cette classe surcharge plusieurs opérateurs, dont l'opérateur d'indexation (`[ ]`), l'opérateur plus (+) et l'opérateur plus-égal (+=). L'opérateur d'indexation est surchargé deux fois : la première comme méthode constante renvoyant un caractère, la deuxième comme méthode variable renvoyant une référence sur un caractère.

La version non constante est utilisée dans des instructions comme celle-ci (ligne 161) :

```
S1[4]='x';
```

Elle permet d'avoir un accès direct aux caractères de la chaîne. Cette version renvoie une référence au caractère afin que la fonction appelante puisse le traiter.

La version constante, quant à elle, permet d'avoir accès aux objets constants de la classe `String`, pour l'implémentation du constructeur de copie, par exemple (à partir de la ligne 63). Vous pouvez noter que le programme lit la variable `rhs[i]`, bien qu'elle soit déclarée comme une `const String &`. Comme il est interdit de lire cet objet à l'aide d'une fonction membre variable, l'opérateur d'indexation doit être surchargé avec une fonction d'accès constante. Si l'objet renvoyé était volumineux, il serait préférable de renvoyer une référence constante, mais c'est inutile ici car un caractère n'occupe qu'un octet en mémoire.

Le constructeur par défaut est implémenté des lignes 34 à 39. Il crée une chaîne de taille nulle ne contenant que le caractère nul final. Ce dernier n'est pas pris en compte par la classe `String` lorsqu'elle renvoie la longueur de la chaîne.

Le constructeur de copie définit la longueur de la nouvelle chaîne, en ajoutant le caractère nul à la longueur existante. Pour cela, il copie un à un les caractères de la chaîne source vers la chaîne cible, puis ajoute le caractère nul (lignes 63 à 70). À la différence des opérateurs d'affectation, les constructeurs de copie n'ont pas besoin de tester si la chaîne copiée dans ce nouvel objet est cet objet lui-même ; cela ne peut jamais arriver.

De la ligne 53 à la ligne 60, un constructeur, similaire au constructeur de copie, prend en paramètre une chaîne de type C. Pour déterminer la taille de la chaîne existante, il appelle la fonction standard `strlen()`.

La ligne 28 déclare le constructeur `String(unsigned short)` comme une méthode privée, afin d'empêcher toute tentative de création d'une chaîne de longueur arbitraire par les clients de la classe. Il a pour rôle d'aider les fonctions spéciales à créer des chaînes comme dans le cas, par exemple, de `operator+=` à la ligne 131. Cette fonctionnalité est décrite dans la section consacrée à l'opérateur `+=`, plus loin dans ce chapitre.

Aux lignes 44 à 50, le constructeur `String(unsigned short)` complète tous les éléments du tableau par des caractères nuls (`\0`). C'est pourquoi la boucle `for` vérifie que le nombre d'éléments lus est bien inférieur ou égal à la longueur de la chaîne (`i< = lg`).

La chaîne de caractères gérée par la classe est supprimée par le destructeur (lignes 73 à 77). Assurez-vous de ne pas oublier les crochets après le mot-clé `delete` car il faut supprimer tous les éléments du tableau, pas seulement le premier.

L'opérateur d'affectation (=) est surchargé aux lignes 81 à 92. Cette méthode détermine si les deux côtés de l'expression sont identiques ou non. S'ils sont différents, la chaîne existante est supprimée, puis remplacée par la nouvelle. Cette méthode renvoie une référence, ce qui permet d'écrire :

```
Chaine1 = Chaine2 = Chaine3;
```

L'opérateur d'indexation est également surchargé. Il l'est d'abord aux lignes 97 à 103, puis à nouveau aux lignes 107 à 113. Dans les deux cas, il effectue un test très simple pour vérifier que l'indice fourni est inférieur à la borne supérieure de la chaîne. Si l'utilisateur tente d'accéder à un caractère en dehors de la limite supérieure du tableau, l'opérateur renvoie le dernier caractère.

L'opérateur + permet de concaténer deux chaînes (lignes 117 à 127) car il est pratique de pouvoir écrire :

```
Chaine3 = Chaine1 + Chaine2;
```

afin que `Chaine3` contienne le résultat de la concaténation des deux autres chaînes. Pour cela, l'opérateur `+` additionne les tailles respectives des deux chaînes, puis appelle le constructeur privé en lui passant cette longueur totale afin qu'il crée une chaîne temporaire `temp` de cette longueur en l'initialisant avec des caractères nuls. Ceux-ci sont ensuite remplacés par les contenus des deux chaînes. La chaîne de gauche (`*this`) est copiée en premier, puis c'est au tour de la chaîne de droite (`rhs`). La première boucle `for` parcourt la chaîne de gauche et ajoute chacun de ses caractères à la nouvelle chaîne. La seconde boucle `for` fait de même pour la chaîne de droite. Vous pouvez constater que les compteurs `i` et `j` s'incrémentent simultanément, car `i` désigne l'indice de la chaîne cible, tandis que `j` désigne l'indice de la chaîne de droite.

À la ligne 127, la chaîne `temp` renvoyée par valeur par l'opérateur plus est affectée à la chaîne située à gauche dans l'affectation (`chaine1`). Aux lignes 131 à 143, l'opérateur `+=` agit directement sur la chaîne existante (c'est-à-dire sur la partie gauche de l'instruction `chaine1 += chaine2`). Il fonctionne comme l'opérateur plus, sauf que la valeur temporaire `temp` est affectée à la chaine courante (`*this = temp`) à la ligne 142.

La fonction `main()` (lignes 143 à 173) sert de programme de test pour la classe. La ligne 147 crée un objet `String` en utilisant le constructeur qui prend en paramètre une chaîne de type C. La ligne suivante affiche son contenu, qu'elle obtient *via* un appel à sa méthode d'accès `GetString()`. La ligne 150 crée une seconde chaîne "à la C", qui est ensuite affectée à la première à la ligne 151. La ligne 153 affiche le résultat de cette affectation et montre que la surcharge de l'opérateur d'affectation a bien fonctionné.

La ligne 154 créé une troisième chaîne C appelée `temp2` et la ligne suivante appelle la fonction `strcpy()` pour lui affecter la chaîne littérale `comment ca va ?`. L'opérateur `+=` est ensuite utiliser pour concaténer le contenu de `temp2` à la chaîne `s1` et le résultat est affiché à la ligne 158.

La ligne 160 lit le cinquième caractère de `s1` à l'aide de l'opérateur d'indexation surchargé et l'affiche. Ce caractère reçoit la nouvelle valeur `x` à la ligne 16. Cette ligne fait appel à l'opérateur d'indexation non constant. La ligne 162 montre que ce caractère a bien été modifié dans la chaîne.

Le programme tente alors de lire un caractère après la fin du tableau. D'après ce qui s'affiche, on peut constater que le caractère qui a été renvoyé est le dernier, comme prévu.

Les lignes 166 et 167 créent deux autres objets `String` et la ligne suivante appelle l'opérateur d'addition surchargé pour les concaténer. Le résultat est affiché en ligne 169.

La ligne 171 crée un nouvel objet `String`, `s4`, qui est initialisé avec une chaîne de type C à la ligne suivante à l'aide de l'opérateur d'affectation surchargé. La ligne 173 affiche le

résultat. Vous pourriez vous demander pourquoi cette affectation d'une chaîne C a été autorisée alors que l'opérateur d'affectation a été défini pour attendre un paramètre de type `String` ?

La réponse est que le compilateur attend effectivement un objet `String`, mais qu'il reçoit un tableau de caractères. Par conséquent, il vérifie s'il peut créer un objet `String` à partir des informations reçues. À la ligne 12, vous avez déclaré un constructeur capable de créer des chaînes à partir de tableaux de caractères : le compilateur crée donc un objet `String` temporaire à partir du tableau de caractères qu'on lui a transmis, puis le passe à l'opérateur d'affectation. Cette opération est appelée *transtypage implicite* ou *promotion*. Si vous n'aviez pas déclaré (ni fourni une implémentation) de constructeur prenant en paramètre un tableau de caractères, cette affectation aurait produit une erreur de compilation.

La classe `String` du Listing 13.14 devient assez robuste. Vous pouvez également constater que ce listing est plus long que les précédents. Heureusement, la bibliothèque C++ standard fournit une classe `String` encore plus robuste, que vous pourrez utiliser en incluant la bibliothèque `<string>`.

Listes chaînées et autres structures

Les tableaux sont comme des Tupperware©. Bien qu'ils puissent contenir de grands volumes de données, leur taille est fixe. Si vous choisissez une boître trop grande, une partie de l'espace ne sera donc pas exploitée. À l'inverse, si elle est trop petite, les aliments déborderont et vous serez bien embêté.

Le Listing 13.9 proposait une solution à ce problème. Toutefois, lorsque l'on commence à utiliser de grands tableaux ou lorsque que l'on souhaite déplacer, effacer ou insérer des entrées d'un tableau, le nombre des allocations et des désallocations peut être assez coûteux.

Les listes chaînées ont été conçues pour résoudre ce type de problèmes. Une liste chaînée est une structure de données composée de conteneurs de petite taille liés entre eux en fonction des besoins. Dans notre exemple, l'objectif est de concevoir une classe contenant un seul objet (`Chat` ou `Rectangle`, par exemple), mais pouvant pointer sur le conteneur suivant. Il suffit alors de mettre en place un chaînage de conteneurs en créant un nouveau conteneur à chaque fois que l'on a besoin de stocker un objet et de faire en sorte de le relier aux autres.

Les listes chaînées sont considérées comme un sujet avancé. Vous trouverez plus d'informations dans l' Annexe E.

Création de classes tableaux

Une classe tableau personnalisée présente de nombreux avantage par rapport aux tableaux prédéfinis. Pour les débutants, elle permet déjà d'éviter les accès en dehors des bornes. En outre, une telle classe peut être dynamique : à sa création, elle ne contiendra qu'un élément et grossira en fonction des besoins au cours du programme.

Il est également possible de trier ou de modifier l'ordre des éléments du tableau. Les variantes suivantes sont celles que l'on utilise le plus souvent :

- **Collection ordonnée.** Les éléments sont triés selon un certain critère.
- **Ensemble.** Chaque élément n'apparaît qu'une fois dans le tableau.
- **Dictionnaire.** Les valeurs sont regroupées par paires, le premier élément servant de clé pour extraire le second élément.
- **Tableau creux.** Les indices permettent d'insérer un très grand nombre d'éléments, mais seuls les valeurs réellement stockées consomment de la mémoire. Si le tableau `Creux[200]` ne contient que cinq valeurs, par exemple, il occupe le même espace mémoire que `Creux[4]`.
- **Sac** (ou *bag*). Collection de données non ordonnée, dont les éléments sont ajoutés et obtenus dans un ordre indéterminé..

En surchargeant l'opérateur d'indexation (`[ ]`), vous pouvez transformer une liste chaînée en une collection ordonnée. En supprimant les doublons, une collection devient un ensemble. Si chaque objet de la liste est associé à deux valeurs (l'une étant la clé de la seconde), vous pouvez créer un dictionnaire ou un tableau creux.



Écrire sa propre classe tableau présente de nombreux avantages par rapport à l'utilisation de tableaux intégrés, mais utiliser les implémentations fournies par la bibliothèque standard pour ces classes (ou des versions analogues) est encore mieux.

Questions-réponses

Q Que contient un élément de tableau non initialisé ?

R L'élément contient ce qui est à cet emplacement mémoire à ce moment-là. Les opérations réalisées sur un tel élément peuvent produire des résultats imprévisibles. Si le compilateur respecte le standard C++, les éléments d'un tableau qui sont statiques et qui ne sont pas des objets non locaux seront initialisés à zéro.

Q Est-il possible de combiner des tableaux ?

R Bien sûr. Avec les tableaux simples, vous pouvez combiner les pointeurs pour former un nouveau tableau plus gros. Avec les chaînes de caractères, vous pouvez utiliser certaines fonctions prédéfinies comme `strcat`.

Q Pourquoi créer une liste chaînée si un tableau fait l'affaire ?

R La taille d'un tableau est définie une fois pour toutes, alors qu'une liste chaînée peut être réduite ou agrandie de façon dynamique pendant l'exécution du programme. L'Annexe E traite de la création de listes chaînées.

Q Pourquoi utiliser les tableaux prédéfinis si l'on peut créer des classes tableaux plus performantes ?

R Les tableaux prédéfinis sont rapides et simples d'emploi et vous en avez généralement besoin pour construire votre propre classe tableau.

Q Existe-t-il une meilleure construction que les tableaux ?

R Au Chapitre 19, vous verrez les modèles et la STL (*Standard Template Library*), qui contient des modèles de collections disposant de toutes les fonctionnalités dont vous avez besoin. L'utilisation de ces modèles est préférable à la création de classes personnelles.

Q Est-ce qu'une classe `string` doit utiliser un `char *` pour stocker le contenu de la chaîne ?

R Non. Elle peut utiliser le moyen de stockage que son concepteur juge le plus adapté.

Testez vos connaissances

1. Citez le premier et le dernier élément de `Tableau[25]`.
2. Comment déclare-t-on un tableau à plusieurs dimensions ?
3. Initialisez les éléments d'un tableau déclaré comme `Tableau[2][3][2]`.
4. Combien y a-t-il d'éléments dans `Tableau[10][5][20]` ?
5. En quoi une liste chaînée diffère-t-elle d'un tableau ?
6. Combien de caractères sont stockés dans la chaîne "Jean apprend C++" ?
7. Quel est le dernier élément de la chaîne "Qui ne dit mot consent" ?

Exercices

1. Déclarez un tableau à deux dimensions représentant un jeu de morpion.
2. Initialisez à zéro tous les éléments du tableau de l'exercice précédent.
3. Écrivez un programme contenant quatre tableaux de caractères. Les trois premiers doivent contenir respectivement votre prénom, vos initiales et votre nom. Utilisez la fonction de copie de chaîne que nous avons présentée dans ce chapitre pour regrouper ces chaînes dans le quatrième tableau, qui contiendra ainsi votre nom complet.
4. CHERCHEZ L'ERREUR dans ce fragment de code :

```
unsigned short Tableau[5][4];
for (int i = 0; i<4; i++)
     for (int j = 0; j<5; j++)
          Tableau[i][j] = i+j;
```

5. CHERCHEZ L'ERREUR dans ce fragment de code :

```
unsigned short Tableau[5][4];
for (int i = 0; i<=5; i++)
     for (int j = 0; j<=4; j++)
          Tableau[i][j] = 0;
```



14

Polymorphisme

Au sommaire de ce chapitre

- Caractéristiques et utilisation de l'héritage multiple
- Nature et rôle de l'héritage virtuel
- Caractéristiques des classes abstraites
- Caractéristiques des fonctions virtuelles pures

Au Chapitre 12, vous avez appris à écrire des fonctions virtuelles dans des classes dérivées. Ces fonctions sont les briques de base du polymorphisme, qui est la capacité de lier des objets d'une classe dérivée à des pointeurs vers la classe de base en cours d'exécution.

Problèmes liés à l'héritage simple

Alors que vous travaillez avec des classes d'animaux, vous décidez de créer une hiérarchie de classes dérivées `Oiseau` et `Mammiferes`. La classe `Oiseau` comprend la fonction membre `Voler()`, alors que la classe `Mammiferes` a été divisée en un certain nombre de

types, parmi lesquels figure le type `Cheval`. La classe `Cheval` inclut les fonctions membres `Hennir()` et `Galoper()`.

Soudain, vous réalisez que vous avez besoin d'un objet `Pegase`, un animal à mi-chemin entre le cheval et l'oiseau. Pégase est capable de voler, de hennir et de galoper. Vous ne pouvez pas résoudre ce problème avec l'héritage simple.

Avec l'héritage simple, vous ne pouvez dériver que d'une seule classe. Vous pouvez créer `Pegase` à partir de la classe `Oiseau`, mais alors il ne saura ni `Hennir()` ni `Galoper()`. Si vous en faites un `Cheval`, il ne saura pas `Voler()`.

Une première solution consiste à copier la méthode `Voler()` dans la classe `Pegase` et à dériver `Pegase` de `Cheval`. Cela fonctionnera correctement, mais au prix de la duplication de la méthode `Voler()` dans les deux classes `Oiseau` et `Pegase`. Si vous modifiez une classe, il ne faudra pas oublier de mettre l'autre à jour. Un développeur se penchant sur votre code plusieurs mois ou années plus tard devra également savoir qu'il doit appliquer ses corrections à deux endroits.

Toutefois, vous rencontrerez rapidement un problème. Si vous créez une liste d'objets `Cheval` et une liste d'objets `Oiseau`, vous devez pouvoir ajouter des objets `Pegase` à l'une ou à l'autre. Mais, si Pégase est un cheval, vous ne pourrez pas l'ajouter à la liste des `Oiseau`.

Vous avez alors deux solutions possibles. Vous pouvez renommer la méthode `Galoper()` en `Deplacer()`, puis redéfinir cette dernière dans l'objet `Pegase` afin qu'elle fasse comme `Voler()`. Il faudra aussi redéfinir la fonction `Deplacer()` dans les autres objets `Cheval` pour qu'elle fasse comme `Galoper()`. Un objet `Pegase` pourrait, par exemple, galoper sur de courtes distances et voler sur des distances plus longues.

```
Pegase::Deplacer(long distance)
{
     if (distance > loin)
          Voler(distance);
     else
          Galoper(distance);
}
```

Ce code est un peu limité, car `Pegase` souhaitera peut-être un jour voler sur une courte distance et galoper sur une distance plus longue. La solution suivante consisterait alors à déplacer la méthode `Voler()` dans la classe `Cheval`, comme le montre le Listing 14.1. Le problème est que la plupart des chevaux ne sachant pas voler, cette méthode ne doit rien faire, sauf si le cheval est un `Pegase`.

Listing 14.1 : Si les chevaux savaient voler…

```
 0:  // Listing 14.1. Si les chevaux savaient voler...
 1:  // Déplacer Voler() dans Cheval
 2:
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  class Cheval
 7:  {
 8:  public:
 9:     void Galoper() { cout << "Au galop...\n"; }
10:     virtual void Voler()
10a:       { cout << "Les chevaux ne savent pas voler.\n" ; }
11:  private:
12:     int sonAge;
13:  };
14:
15:  class Pegase : public Cheval
16:  {
17:  public:
18:     virtual void Voler()
19:     { cout<<"Je vole ! Je vole ! Je vole!\n"; }
20:  };
21:
22:  const int NbChevaux = 5;
23:  int main()
24:  {
25:     Cheval* Ranch[NbChevaux];
26:     Cheval* pCheval;
27:     int choix, i;
28:     for (i = 0; i< NbChevaux; i++)
29:     {
30:        cout << "(1)Cheval (2)Pegase - Votre choix : ";
31:        cin >> choix;
32:        if (choix == 2)
33:           pCheval = new Pegase;
34:        else
35:           pCheval = new Cheval;
36:        Ranch[i] = pCheval;
37:     }
38:     cout << endl;
39:     for (i = 0; i < NbChevaux; i++)
40:     {
41:        Ranch[i]->Voler();
42:        delete Ranch[i];
43:     }
44:     return 0;
45:  }
```

Ce programme produit le résultat suivant :

```
(1)Cheval (2)Pegase - Votre choix : 1
(1)Cheval (2)Pegase - Votre choix : 2
(1)Cheval (2)Pegase - Votre choix : 1
(1)Cheval (2)Pegase - Votre choix : 2
(1)Cheval (2)Pegase - Votre choix : 1

Les chevaux ne savent pas voler.
Je vole ! Je vole ! Je vole !
Les chevaux ne savent pas voler.
Je vole ! Je vole ! Je vole !
Les chevaux ne savent pas voler.
```

Ce programme s'exécute normalement, à condition que la classe `Cheval` comprenne une méthode `Voler()`.

La ligne 10 ajoute la méthode `Voler()` à `Cheval`. Dans une vraie classe, vous pourriez faire en sorte qu'elle produise une erreur ou échoue en silence. À la ligne 18, la classe `Pegase` redéfinit la méthode `Voler()` pour qu'elle produise un message enthousiaste.

Le tableau des pointeurs `Cheval`, appelé `Ranch`, permet de démontrer à la ligne 25 que la méthode `Voler()` appropriée est appelée grâce la liaison dynamique d'un objet `Cheval` ou d'un objet `Pegase`.

Aux lignes 28 à 37, l'utilisateur doit choisir un `Cheval` ou un `Pegase`. Un objet du type correspondant est créé et placé dans le tableau `Ranch`.

Aux lignes 38 à 43, le programme parcourt à nouveau le tableau `Ranch`, cette fois-ci pour appeler la méthode `Voler()` sur chaque élément du tableau. En lisant les lignes produites par le programme, vous pouvez constater que la méthode `Voler`() appelée dépend du fait que l'objet soit un `Cheval` ou un `Pegase`. Ce programme n'utilisant plus les objets de `Ranch`, la ligne 42 appelle `delete` pour libérer la mémoire qu'ils occupaient.



Ces exemples ont été réduits à l'essentiel afin de mettre l'accent sur le point qui nous intéresse. Les constructeurs, les destructeurs virtuels entre autres, ne figurent pas ici. Cette pratique n'est pas conseillée pour vos programmes.

Filtrage ascendant

Placer la fonction requise à un niveau supérieur dans la hiérarchie des classes constitue la solution la plus courante à ce problème et produit de nombreuses fonctions qui "remontent" dans la classe de base. Cette dernière court alors le risque d'être transformée en un espace de noms global pour toutes les fonctions qui pourraient servir à l'une des classes

dérivées. La taille de la classe de base devient alors démesurée et l'approche orientée objet du programme est sérieusement compromise, ce qui est dommage lorsqu'on utilise un langage orienté objet comme C++.

En général, on souhaite faire remonter vers le haut de la hiérarchie une fonctionnalité partagée, sans déplacer l'interface de chaque classe. Si deux classes ont en commun une classe de base (comme c'est le cas des classes dérivées `Cheval` et `Oiseau` avec la classe `Animal`) et une méthode en commun (les oiseaux et les chevaux mangent, par exemple), il est souhaitable de déplacer cette fonctionnalité vers la classe de base et d'en faire une méthode virtuelle.

Ce qu'il faut éviter, par contre, c'est de remonter une méthode comme `Voler()` à un endroit où elle n'a rien à faire uniquement pour pouvoir l'appeler dans certaines classes dérivées.

Transtypage descendant

Une autre possibilité, toujours dans le cadre de l'héritage simple, consisterait à conserver la méthode `Voler()` dans l'objet `Pegase` et de l'appeler uniquement lorsque le pointeur pointe effectivement sur un objet `Pegase`. Vous devrez pour cela être capable d'interroger le pointeur pour identifier le type pointé. Cette identification du type au moment de l'exécution est nommée RTTI (*Run Time Type Identification*).

RTTI étant une nouvelle spécification C++, les compilateurs ne le prennent pas tous en charge. Si votre compilateur ne supporte pas RTTI, vous pouvez imiter son fonctionnement en plaçant une méthode qui renvoie un type énuméré dans chaque classe. Vous aurez ainsi la possibilité de tester le type au moment de l'exécution et d'appeler `Voler()` s'il a renvoyé `Pegase`.



N'abusez pas de RTTI dans vos classes. Son emploi pourrait être le signe d'une mauvaise conception. Envisagez plutôt l'utilisation des méthodes virtuelles, des modèles ou de l'héritage multiple.

Dans l'exemple précédent, les objets `Cheval` et `Pegase` ont été déclarés et placés dans un tableau d'objets `Cheval`. Avec RTTI, il faudrait vérifier chacun de ces éléments du tableau pour savoir s'il s'agit d'un cheval ou d'un `Pegase`.

Pour appeler `Voler()`, cependant, il faut transtyper le type du pointeur pour l'informer que l'objet qu'il pointe est un `Pegase` et non un `Cheval`. C'est ce que l'on appelle un *transtypage descendant* car on transtype d'un type plus général vers un type plus spécialisé.

Officiellement, mais parfois à contrecœur, C++ gère ces transtypages à l'aide du nouvel opérateur `dynamic_cast`.

Si vous affectez à un pointeur d'une classe de base, comme `Cheval`, un pointeur d'une classe dérivée comme `Pegase`, vous pouvez utiliser le pointeur sur `Cheval` de façon polymorphe. Si vous avez besoin d'accéder à l'objet `Pegase`, il suffit de créer un pointeur sur `Pegase` et d'utiliser l'opérateur `dynamic_cast` pour effectuer cette conversion.

Lors de l'exécution, le pointeur de base sera examiné et, si la conversion est correcte, votre nouveau pointeur sur `Pegase` sera opérationnel. Si cette conversion est incorrecte, par contre, vous ne disposeriez pas d'un objet `Pegase` et ce nouveau pointeur vaudrait alors null. Le Listing 14.2 illustre ce processus.

Listing 14.2 : Transtypage descendant

```
 0:  // Listing 14.2 Utilisation de transtypage descendant
 1:  // dynamique : fonctionnalité RTTI
 2:
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  enum TYPE { CHEVAL, PEGASE };
 7:
 8:  class Cheval
 9:  {
10:  public:
11:     virtual void Galoper(){ cout << "Au galop...\n"; }
12:
13:  private:
14:     int sonAge;
15:  };
16:
17:  class Pegase : public Cheval
18:  {
19:  public:
20:      virtual void Voler()
21:         {cout<<"Je vole ! Je vole ! Je vole !\n";}
22:  };
23:
24:  const int NbChevaux = 5;
25:  int main()
26:  {
27:     Cheval* Ranch[NbChevaux];
28:     Cheval* pCheval;
29:     int choix, i;
```

```
30:     for (i = 0; i < NbChevaux; i++)
31:     {
32:        cout << "(1)Cheval (2)Pegase - Votre choix : ";
33:        cin >> choix;
34:        if (choix == 2)
35:           pCheval = new Pegase;
36:        else
37:           pCheval = new Cheval;
38:        Ranch[i] = pCheval;
39:     }
40:     cout << endl;
41:     for (i = 0; i < NbChevaux; i++)
42:     {
43:        Pegase *pPeg = dynamic_cast< Pegase *> (Ranch[i]);
44:        if (pPeg != NULL)
45:           pPeg->Voler();
46:        else
47:           cout << "Ce n'est qu'un cheval\n";
48:
49:        delete Ranch[i];
50:     }
51:     return 0;
52:  }
```

Ce programme produit le résultat suivant :

```
(1)Cheval (2)Pegase - Votre choix : 1
(1)Cheval (2)Pegase - Votre choix : 2
(1)Cheval (2)Pegase - Votre choix : 1
(1)Cheval (2)Pegase - Votre choix : 2
(1)Cheval (2)Pegase - Votre choix : 1

Ce n'est qu'un cheval
Je vole ! Je voler ! Je vole !
Ce n'est qu'un cheval
Je vole ! Je voler ! Je vole !
Ce n'est qu'un cheval
```

Avertissement du compilateur

Le compilateur Microsoft Visual C++ peut produire l'avertissement suivant :

```
warning C4541: 'dynamic_cast' used on polymorphic type 'class Cheval'
with /GR-; unpredictable behavior may result.
```

Lorsque j'exécute le programme, un message indique une fin d'exécution inhabituelle et me demande de contacter l'équipe d'assistance.

Ces messages sont assez confus. Vous pouvez résoudre le problème de la façon suivante :

1. Dans votre projet, choisissez Projet/Configuration.
2. Choisissez l'onglet C/C++.
3. Sélectionnez C++ dans la liste déroulante Catégorie.
4. Cochez la case Activer Runtime Type Information (RTTI).
5. Recréez la totalité du projet.

Si vous utilisez le compilateur Visual C++en ligne de commande, ajoutez l'option `/GR` :

```
Cl /GR List1402.cpp
```

Cette solution fonctionne aussi, mais elle n'est pas conseillée. La fonction `Voler()` reste en dehors de la classe `Cheval` et n'est pas appelée avec les objets de cette classe. Lorsqu'elle est invoqué sur des objets `Pegase` (ligne 45), ceux-ci doivent être convertis explicitement (ligne 43). Les objets `Cheval` ne disposant pas de la méthode `Voler()`, il faut indiquer au pointeur que l'objet pointé est un `Pegase` avant de l'utiliser.

La nécessité de convertir l'objet `Pegase` est probablement le signe d'une mauvaise conception. En fait, ce programme compromet le polymorphisme des fonctions virtuelles puisqu'il dépend du transtypage de l'objet vers son type effectif au moment de l'exécution.

Ajouter dans deux listes

Ces solutions présentent un autre inconvénient : si vous avez déclaré que `Pegase` est une classe dérivée de `Cheval`, vous ne pouvez pas ajouter un objet `Pegase` à une liste d'objets `Oiseau`. Vous avez déjà été contraint soit d'introduire `Voler()` dans la classe `Cheval`, soit de transtyper le pointeur vers le bas et vous ne disposez toujours pas de la fonctionnalité recherchée.

Il existe une dernière solution avec l'héritage simple. Vous pouvez remonter les trois méthodes `Voler()`, `Hennir()` et `Galoper()` dans une classe de base commune à `Oiseau` et `Cheval` : la classe `Animal`. Désormais, au lieu d'avoir une liste d'oiseaux et une liste de chevaux, vous pouvez donc utiliser une unique liste d'animaux. Cela fonctionne, mais vous finirez par avoir une classe de base qui possède toutes les caractéristiques de toutes ses classes descendantes. Pourquoi alors créer ces dernières ?

Bien sûr, vous pourriez laisser ces méthodes où elles sont et effectuer un transtypage descendant des objets `Cheval`, `Oiseau` et `Pegase`, mais c'est encore pire !

Faire	*Ne pas faire*
● Déplacer la fonctionnalité vers le haut dans une hiérarchie d'héritage lorsque que cela est cohérent avec la signification de la classe ancêtre. ● Éviter les opérations qui dépendent du type de l'objet au moment de l'exécution – faire plutôt appel aux méthodes virtuelles, aux modèles et à l'héritage multiple.	● Encombrer les classes ancêtres avec des fonctionnalités qui ne sont ajoutées que pour soutenir le besoin de polymorphisme dans les classes descendantes. ● Convertir des pointeurs sur des objets de la classe de base en pointeurs vers des objets des classes dérivées.

Héritage multiple

L'héritage multiple consiste à dériver une nouvelle classe à partir de plusieurs classes de base. Pour dériver plusieurs classes de base, vous devez placer une virgule entre chacune d'elle dans la liste d'héritage :

```
class ClasseDerivee : public ClasseBase1, public ClasseBase2 {}
```

Cette ligne ressemble exactement à une déclaration d'héritage simple, mais avec une classe de base supplémentaire, *`ClasseBase2`*.

Dans le Listing 14.3, la classe `Pegase` dérive maintenant à la fois de la classe `Cheval` et de la classe `Oiseau`. Le programme ajoute ensuite des objets `Pegase` à des listes de ces deux types.

Listing 14.3 : Héritage multiple

```
 0:  // Listing 14.3 - Héritage multiple
 1:
 2:  #include <iostream>
 3:  using std::cout;
 4:  using std::cin;
 5:  using std::endl;
 6:
 7:  class Cheval
 8:  {
 9:  public:
10:     Cheval() { cout << "Constructeur de Cheval... "; }
11:     virtual ~Cheval() { cout << "Destructeur de Cheval... "; }
12:     virtual void Hennir() const { cout << "Hihiii !... "; }
13:  private:
14:     int sonAge;
15:  };
```

```
16:
17:  class Oiseau
18:  {
19:  public:
20:     Oiseau() { cout << "Constructeur de Oiseau... "; }
21:     virtual ~Oiseau() { cout << "Destructeur de Oiseau... "; }
22:     virtual void Gazouiller() const { cout << "Cuicui... ";  }
23:     virtual void Voler() const
24:     {
25:        cout <<
25a:         "Je vole ! Je vole ! Je vole ! ";
26:     }
27:  private:
28:     int sonPoids;
29:  };
30:
31:  class Pegase : public Cheval, public Oiseau
32:  {
33:  public:
34:     void Gazouiller() const { Hennir(); }
35:     Pegase() { cout << "Constructeur de Pegase... "; }
36:     ~Pegase() { cout << "Destructeur de Pegase...  "; }
37:  };
38:
39:  const int Max = 2;
40:  int main()
41:  {
42:     Cheval* Ranch[Max];
43:     Oiseau* Voliere[Max];
44:     Cheval * pCheval;
45:     Oiseau * pOiseau;
46:     int choix, i;
47:     for (i = 0; i < Max; i++)
48:     {
49:        cout << "\n(1)Cheval (2)Pegase - Votre choix : ";
50:        cin >> choix;
51:        if (choix == 2)
52:           pCheval = new Pegase;
53:        else
54:           pCheval = new Cheval;
55:        Ranch[i] = pCheval;
56:     }
57:     for (i = 0; i < Max; i++)
58:     {
59:        cout << "\n(1)Oiseau (2)Pegase - Votre choix : ";
60:        cin >> choix;
61:        if (choix == 2)
```

```
62:           pOiseau = new Pegase;
63:        else
64:           pOiseau = new Oiseau;
65:        Voliere[i] = pOiseau;
66:     }
67:
68:     cout << endl;
69:     for (i = 0; i < Max; i++)
70:     {
71:        cout << "\nRanch[" << i << "] : " ;
72:        Ranch[i]->Hennir();
73:        delete Ranch[i];
74:     }
75:
76:     for (i = 0; i < Max; i++)
77:     {
78:        cout << "\nVoliere[" << i << "] : " ;
79:        Voliere[i]->Gazouiller();
80:        Voliere[i]->Voler();
81:        delete Voliere[i];
82:     }
83:     return 0;
84:  }
```

Ce programme produit le résultat suivant :

```
(1)Cheval (2)Pegase - Votre choix : 1
Constructeur de Cheval...
(1)Cheval (2)Pegase - Votre choix : 2
Constructeur de Cheval... Constructeur de Oiseau... Constructeur de Pegase
(1)Oiseau (2)Pegase - Votre choix : 1
Constructeur de Oiseau...
(1)Oiseau (2)Pegase - Votre choix : 2
Constructeur de Cheval... Constructeur de Oiseau... Constructeur de Pegase
Ranch[0] : Hihiii !... Destructeur de Cheval...
Ranch[1] : Hihiii !... Destructeur de Pegase.. Destructeur de Oiseau...
Destructeur de Cheval...
Voliere[0] : Cuicui... Je vole ! Je vole ! Je vole ! Destructeur de Oiseau...
Voliere[1]: Hihiii!... Je vole ! Je vole ! Je vole !
Destructeur de Pegase... Destructeur de Oiseau... Destructeur de Cheval...
```

La classe `Cheval` est déclarée aux lignes 7 à 15. Le constructeur et le destructeur affichent un message et la méthode `Hennir()` donne la parole à notre pégase.

La classe `Oiseau` est déclarée aux lignes 17 à 29. Outre un constructeur et un destructeur, elle inclut deux méthodes : `Gazouiller()` et `Voler()`, qui produisent toutes les deux un

message caractéristique. Dans un vrai programme, elles pourraient produire un son ou lancer une animation multimédia.

Enfin, la classe `Pegase` est déclarée aux lignes 31 à 37. Elle utilise la syntaxe que nous avons présentée pour l'héritage multiple. Elle dérive à la fois de la classe `Cheval` et de la classe `Oiseau` (ligne 31). Cette classe redéfinit la méthode `Gazouiller()` (ligne 34) pour appeler simplement la méthode `Hennir()`, héritée de la classe `Cheval`.

La fonction principale de ce programme crée deux listes : la liste `Ranch`, qui contient des pointeurs sur des objets `Cheval` (ligne 42) et la liste `Voliere` qui contient des pointeurs sur des objets `Oiseau` (ligne 43). Les lignes 47 à 56 ajoutent des objets `Cheval` et `Pegase` à la liste `Ranch` alors que les lignes 57 à 66 ajoutent des objets `Oiseau` et `Pegase` à `Voliere`.

Les appels des méthodes virtuelles sur des pointeurs de `Oiseau` et de `Cheval` permettent de réaliser les opérations appropriées sur les objets `Pegase`. À la ligne 79, par exemple, les éléments du tableau `Voliere` appellent la méthode `Gazouiller()` sur les objets sur lesquels ils pointent, car la classe `Oiseau` a déclaré cette méthode comme virtuelle.

À chaque fois qu'un objet `Pegase` est créé, l'opération se répercute à la fois dans sa partie `Oiseau` et sa partie `Cheval` (comme le montre le résultat). De même, lorsqu'un objet `Pegase` est détruit, ses parties `Oiseau` et `Cheval` sont également supprimées grâce aux destructeurs virtuels.

Déclaration d'héritage multiple

Pour qu'une classe hérite de une ou plusieurs classes, il faut fournir la liste de ses classes de bases après le caractère deux-points qui suit le nom de la nouvelle classe. Les différentes classes de base doivent être séparées par des virgules.

Exemple 1

```
class Pegase : public Cheval, public Oiseau
```

Exemple 2

```
class Bardot : public Cheval, public Anesse
```

Parties d'un objet à héritage multiple

Lorsqu'un objet `Pegase` est créé en mémoire, les deux classes de base font partie de cet objet, comme le montre la Figure 14.1. Cette figure représente un objet `Pegase` entier. Celui-ci contient les méthodes ajoutées à la classe `Pegase` et celles héritées des classes de base.

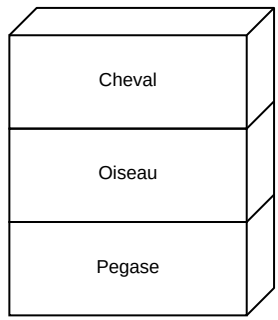


Figure 14.1
Objet d'un héritage multiple.

Les objets associés à plusieurs classes de base posent un certain nombre de problèmes. Par exemple, que se passe-t-il si deux classes de base ont des données ou des méthodes virtuelles portant le même nom ? Comment sont initialisés les différents constructeurs des classes de base ? Que se passe-t-il si plusieurs classes de base héritent de la même classe ? Les sections qui suivent répondent à ses questions et expliquent comment mettre en œuvre l'héritage multiple.

Constructeurs d'objets de l'héritage multiple

Si `Pegase` dérive de `Cheval` et d'`Oiseau` et que chacune de ces classes de base fournit des constructeurs attendant des paramètres, la classe dérivée va initialiser ces constructeurs tour à tour, comme le montre le Listing 14.4.

Listing 14.4 : Appel de plusieurs constructeurs

```
 0:  // Listing 14.4
 1:  // Appel de plusieurs constructeurs
 2:
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  typedef int TAILLE;
 7:  enum COULEUR { Rouge, Vert, Bleu, Jaune, Blanc, Noir, Brun } ;
 8:
 9:  class Cheval
10:  {
11:  public:
12:     Cheval(COULEUR couleur, TAILLE taille);
13:     virtual ~Cheval() { cout << "Destructeur de Cheval...\n"; }
14:     virtual void Hennir()const { cout << "Hihiii !... "; }
15:     virtual TAILLE GetTaille() const { return saTaille; }
16:     virtual COULEUR GetCouleur() const { return saCouleur; }
17:  private:
```

```
18:     TAILLE saTaille;
19:     COULEUR saCouleur;
20:  };
21:
22:  Cheval::Cheval(COULEUR couleur, TAILLE taille):
23:    saCouleur(couleur),saTaille(taille)
24:  {
25:     cout << "Constructeur de Cheval...\n";
26:  }
27:
28:  class Oiseau
29:  {
30:  public:
31:     Oiseau(COULEUR couleur, bool migrates);
32:     virtual ~Oiseau()
32a:     {cout << "Destructeur de Oiseau...\n";  }
33:     virtual void Gazouiller()const { cout << "Cuicui... ";  }
34:     virtual void Voler()const
35:     {
36:        cout <<
36a:         "Je vole ! Je vole ! Je vole ! ";
37:     }
38:     virtual COULEUR GetCouleur()const { return saCouleur; }
39:     virtual bool GetMigration() const { return saMigration; }
40:
41:  private:
42:     COULEUR saCouleur;
43:     bool saMigration;
44:  };
45:
46:  Oiseau::Oiseau(COULEUR couleur, bool migrateur):
47:    saCouleur(couleur), saMigration(migrates)
48:  {
49:     cout << "Constructeur de Oiseau...\n";
50:  }
51:
52:  class Pegase : public Cheval, public Oiseau
53:  {
54:  public:
55:     void Gazouiller()const { Hennir(); }
56:     Pegase(COULEUR, TAILLE, bool,long);
57:     ~Pegase() {cout << "Destructeur de Pegase...\n";}
58:     virtual long GetNbCroyants() const
59:     {
60:        return  nbCroyants;
61:     }
62:
```

```
63:  private:
64:     long nbCroyants;
65:  };
66:
67:  Pegase::Pegase(
68:     COULEUR uneCouleur,
69:     TAILLE unetaille,
70:     bool migrateur,
71:     long nb):
72:     Cheval(uneCouleur, uneTaille),
73:     Oiseau(uneCouleur, migrateur),
74:     nbCroyants(nb)
75:  {
76:  cout << "Constructeur de Pegase...\n";
77:  }
78:
79:  int main()
80:  {
81:     Pegase *pPeg = new Pegase(Rouge, 5, true, 10);
82:     pPeg->Voler();
83:     pPeg->Hennir();
84:     cout << "\nPegase mesure " << pPeg->GetTaille();
85:     cout << " cm et ";
86:     if (pPeg->GetMigration())
87:        cout << "il peut migrer.";
88:     else
89:        cout << "il ne peut pas migrer.";
90:     cout << "\nAu total " << pPeg->GetNbCroyants();
91:     cout << " personnes croient qu'il existe." << endl;
92:     delete pPeg;
93:     return 0;
94:  }
```

Ce programme produit le résultat suivant :

```
Constructeur de Cheval...
Constructeur de Oiseau...
Constructeur de Pegase...
Je vole ! Je vole ! Je vole ! Hihiii !...
Pegase mesure 5 cm et il peut migrer.
Au total, 10 personnes croient qu'il existe.

Destructeur de Pegase...
Destructeur de Oiseau...
Destructeur de Cheval...
```

La classe `Cheval` est déclarée aux lignes 9 à 20. Le constructeur prend deux paramètres : une énumération de couleurs déclarée à la ligne 7 et une valeur du type `typedef` déclaré à la ligne 6. Le constructeur initialise simplement les variables membres et affiche un message (ligne 21 à 25).

La classe `Oiseau` est déclarée de la ligne 28 à la ligne 44 et l'implémentation de son constructeur se trouve aux lignes 46 à 50. La classe de base `Oiseau` attend deux paramètres, comme celui de la classe `Cheval`. Il faut noter que constructeur de `Cheval` a un paramètre couleur (qui indique la couleur de sa robe), comme le constructeur de `Oiseau` (pour indiquer la couleur de ses plumes). Comme nous le verrons plus loin, ceci peut poser problème lorsque vous souhaiterez demander la couleur de `Pegase`.

La classe `Pegase` est déclarée aux lignes 52 à 65, son constructeur aux lignes 67 à 77. L'initialisation de l'objet `Pegase` comprend trois instructions. Le constructeur de `Cheval` est d'abord initialisé avec une couleur et une taille (ligne 72). Le constructeur de `Oiseau` l'est avec une couleur et une valeur booléenne indiquant s'il s'agit d'un oiseau migrateur (ligne 73). Enfin, la variable membre de `Pegase`, `nbCroyants` est initialisée. Le corps du constructeur de `Pegase` peut alors s'exécuter.

Dans la fonction `main()`, un pointeur sur `Pegase` (ligne 81) permet d'accéder aux fonctions membres héritées des classes de base. L'accès à ces méthodes est relativement évident.

Résolution des ambiguïtés

Dans le Listing 14.4, les deux classes de base `Cheval` et `Oiseau` ont toutes les deux une méthode membre `GetCouleur()`. Vous remarquerez que nous n'avons pas appelé ces méthodes dans le Listing 14.4 ! Si vous souhaitez connaître la couleur de l'objet `Pegase`, un problème va se poser puisque la classe dérivée hérite des classes `Oiseau` et `Cheval`. Toutes les deux ont une couleur et leurs méthodes pour l'obtenir ont un nom et une signature identiques. Le compilateur est donc incapable de résoudre l'ambiguïté si vous ne corrigez pas le code source.

L'instruction :

```
COULEUR saCouleur = pPeg->GetCouleur();
```

provoque une erreur de compilation :

```
Member is ambiguous: 'Cheval::GetCouleur' and 'Oiseau::GetCouleur'
```

Pour résoudre ce problème, vous devez rendre plus explicite l'appel de la méthode :

```
COULEUR saCouleur = pPeg->Cheval::GetCouleur();
```

À chaque fois que vous devez connaître la classe d'une méthode ou une donnée membre dont vous héritez, vous pouvez qualifier cet appel en préfixant le nom du membre par celui de la classe dont il provient.

Si cette méthode avait été redéfinie par la classe `Pegase`, le problème se serait déplacé vers la méthode membre de `Pegase` :

```
virtual COULEUR GetCouleur()const { return Cheval::saCouleur; }
```

Ceci cache le problème aux clients de la classe `Pegase` en encapsulant la connaissance de la classe de base dont on hérite la couleur. Cela dit, un client peut toujours écrire :

```
COULEUR saCouleur = pPeg->Oiseau::GetCouleur();
```

Héritage d'une classe de base commune

Que se passerait-il si `Oiseau` et `Cheval` héritaient toutes les deux de la classe de base `Animal` ? La Figure 14.2 illustre cette relation.

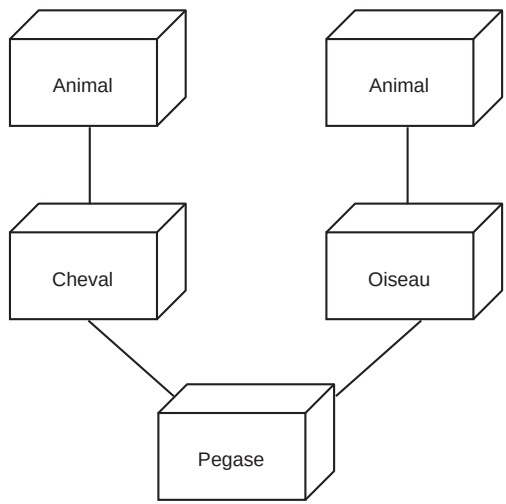


Figure 14.2
Classes de base communes.

Vous pouvez constater qu'il y a maintenant deux objets de la classe de base. Lors de l'appel d'une méthode ou d'une donnée membre de cette classe partagée, une autre ambiguïté survient alors. Si la classe `Animal` contient une variable membre `sonAge` et la méthode membre `GetAge()`, et que vous appelez :

```
pPeg->GetAge()
```

voulez-vous dire que vous appelez la méthode `GetAge()` héritée de `Animal` par l'intermédiaire de `Cheval` ou par l'intermédiaire de `Oiseau` ? Vous devez également résoudre cette ambiguïté, comme on le montre dans le Listing 14.5.

Listing 14.5 : Classes de base communes

```
 0:  // Listing 14.5
 1:  // classes de base communes
 2:
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  typedef int TAILLE;
 7:  enum COULEUR { Rouge, Vert, Bleu, Jaune, Blanc, Noir, Brun } ;
 8:
 9:  class Animal      // classe de base commune Cheval et Oiseau
10:  {
11:  public:
12:     Animal(int);
13:     virtual ~Animal() { cout << "Destructeur de Animal...\n"; }
14:     virtual int GetAge() const { return sonAge; }
15:     virtual void SetAge(int age) { sonAge = age; }
16:  private:
17:     int sonAge;
18:  };
19:
20:  Animal::Animal(int age):
21:    sonAge(age)
22:  {
23:     cout << "Constructeur de Animal...\n";
24:  }
25:
26:  class Cheval : public Animal
27:  {
28:  public:
29:     Cheval(COULEUR couleur, TAILLE taille, int age);
30:     virtual ~Cheval()
30a:       { cout << "Destructeur de Cheval...\n"; }
31:     virtual void Hennir()const { cout << "Hihiii !... "; }
32:     virtual TAILLE GetTaille() const { return saTaille; }
33:     virtual COULEUR GETCouleur() const { return saCouleur; }
34:  protected:
35:     TAILLE saTaille;
36:     COULEUR saCouleur;
37:  };
38:
```

```
39:  Cheval::Cheval(COULEUR couleur, TAILLE taille, int age):
40:    Animal(age),
41:    saCouleur(couleur),saTaille(taille)
42:  {
43:     cout << "Constructeur de Cheval...\n";
44:  }
45:
46:  class Oiseau : public Animal
47:  {
48:  public:
49:     Oiseau(COULEUR couleur, bool migrateur, int age);
50:     virtual ~Oiseau() {cout << "Destructeur de Oiseau...\n";  }
51:     virtual void Gazouiller()const { cout << "CuiCui... ";  }
52:     virtual void Voler()const
53:        { cout << "Je vole ! Je vole ! Je vole ! "; }
54:     virtual COULEUR GetCouleur()const { return saCouleur; }
55:     virtual bool GetMigration() const { return saMigration; }
56:  protected:
57:     COULEUR saCouleur;
58:     bool saMigration;
59:  };
60:
61:  Oiseau::Oiseau(COULEUR couleur, bool migrateur, int age):
62:    Animal(age),
63:    saCouleur(couleur), saMigration(migrateur)
64:  {
65:     cout << "Constructeur de Oiseau...\n";
66:  }
67:
68:  class Pegase : public Cheval, public Oiseau
69:  {
70:  public:
71:     void Gazouiller()const { Hennir(); }
72:     Pegase(COULEUR, TAILLE, bool, long, int);
73:     virtual ~Pegase() {cout << "Destructeur de Pegase...\n";}
74:     virtual long GetNbCroyants() const
75:        { return  nbCroyants; }
76:     virtual COULEUR GetCouleur()const
76a:       { return Cheval::wCouleur; }
77:     virtual int GetAge() const { return Cheval::GetAge(); }
78:  private:
79:     long nbCroyants;
80:  };
81:
82:  Pegase::Pegase(
83:     COULEUR uneCouleur,
84:     HANDS uneTaille,
```

```
 85:     bool migrateur,
 86:     long nb,
 87:     int age):
 88:       Cheval(uneCouleur, uneTaille, age),
 89:       Oiseau(uneCouleur, migrateur, age),
 90:       nbCroyants(nb)
 91:  {
 92:     cout << "Constructeur de Pegase...\n";
 93:  }
 94:
 95:  int main()
 96:  {
 97:     Pegase *pPeg = new Pegase(Rouge, 5, true, 10, 2);
 98:     int age = pPeg->GetAge();
 99:     cout << "Pegase a " << age << " ans.\n";
100:    delete pPeg;
101:    return 0;
102:  }
```

Ce programme produit le résultat suivant :

```
Constructeur de Animal...
Constructeur de Cheval...
Constructeur de Animal...
Constructeur de Oiseau...
Constructeur de Pegase...
Pegase a 2 ans.
Destructeur de Pegase...
Destructeur de Oiseau...
Destructeur de Animal...
Destructeur de Cheval...
Destructeur de Animal...
```

Les fonctionnalités de ce programme sont intéressantes à bien des égards. Déclarée de la ligne 9 à la ligne 18, la classe `Animal` contient une variable membre `sonAge` et deux méthodes d'accès, `GetAge()` et `SetAge()`.

À la ligne 26, la classe `Cheval` est déclarée comme classe dérivée de `Animal`. Son constructeur passe le troisième paramètre, `age`, à la classe de base (ligne 40). La classe `Cheval` ne redéfinit pas la fonction `GetAge()`, mais se contente d'en hériter.

À la ligne 46, la classe `Oiseau` est déclarée comme classe dérivée de `Animal`. Son constructeur initialise la classe de base (ligne 68) avec le paramètre `age`. Comme pour la classe dérivée précédente, la classe `Oiseau` hérite de la fonction `GetAge()` sans la redéfinir.

`Pegase` hérite à la fois de `Oiseau` et de `Animal`, ce qui crée une chaîne d'héritage avec deux fois la classe `Animal`. Si vous appelez la fonction `GetAge()` sur un objet `Pegase`

alors qu'elle n'a pas été redéfinie dans cette classe, il faudrait indiquer clairement le nom de la classe dont elle dépend afin de supprimer toute ambiguïté.

Ce problème est résolu à la ligne 77, lorsque l'objet `Pegase` redéfinit la méthode `GetAge()` en effectuant simplement un *chaînage*, c'est-à-dire un appel de la même méthode dans la classe de base.

Un *chaînage ascendant* permet d'atteindre deux objectifs : supprimer toute ambiguïté entre les deux classes de base ou effectuer un traitement puis demander à la méthode de la classe de base d'en faire de même. Parfois, ce traitement peut s'effectuer avant le chaînage, parfois le chaînage peut avoir lieu avant le traitement local : cela dépend de vos besoins.

Le constructeur de `Pegase`, qui commence à la ligne 82, attend cinq paramètres : la couleur de l'animal, sa taille, s'il est migrateur, le nombre de personnes qui croient en son existence et son âge. À la ligne 88, il initialise la partie héritée de la classe `Cheval` avec la couleur, la taille et l'âge. Il ajoute ensuite la couleur, la migration et l'âge appartenant à la partie `Oiseau` (ligne 89). Enfin, il initialise sa propre variable `nbCroyants` à la ligne 90.

L'appel du constructeur de `Cheval` à la ligne 88 invoque l'implémentation située à la ligne 39. Il affecte à l'objet `Pegase` la valeur d'âge commune à `Animal` et à `Cheval`, puis initialise les variables membres `saCouleur` et `saTaille` de `Cheval`.

L'appel du constructeur de `Oiseau` à la ligne 89 invoque l'implémentation située à la ligne 46. Ici aussi, le paramètre `age` sert à initialiser la variable membre de la partie `Animal` de `Oiseau.`

Vous remarquerez que le paramètre de couleur associé à `Pegase` permet d'initialiser les variables membres dans `Oiseau` et `Cheval`. Vous constaterez également que le paramètre `age` sert à initialiser la variable `sonAge` dans la classe de base `Animal` de `Cheval` et dans la classe de base `Animal` de `Oiseau`.



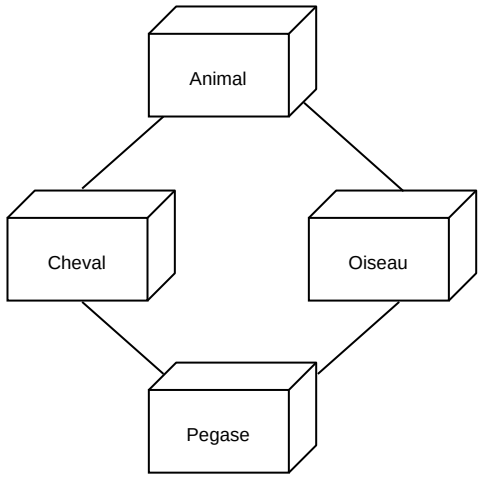
Dès que vous levez explicitement l'ambiguïté d'une classe ancêtre, il existe un risque qu'une nouvelle classe insérée entre votre classe et son ancêtre amène votre classe à appeler "après" le nouvel ancêtre dans l'ancien ancêtre. Cela peut avoir des effets inattendus.

Héritage virtuel

Dans le listing précédent, la classe `Pegase` faisait beaucoup d'efforts pour lever toutes les ambiguïtés concernant les versions de la classe de base `Animal` qu'elle utilisait. La plupart du temps, cette décision est arbitraire – après tout, les classes `Cheval` et `Oiseau` ont la même classe de base.

Comme le montre la Figure 14.3, C++ vous permet de préciser que vous ne voulez pas deux copies de la classe de base partagée, comme dans la Figure 14.2, mais une seule.

Figure 14.3
Héritage en losange.



Pour ce faire, il faut que `Animal` soit une classe de base *virtuelle* de `Cheval` et `Oiseau`. La classe `Animal` ne sera en rien modifiée ; `Cheval` et `Oiseau` utiliseront le mot-clé `virtual` dans leur déclarations d'héritage. La classe `Pegase`, par contre, subit des modifications non négligeables.

En général, un constructeur d'objet initialise uniquement ses propres variables membres et sa classe de base. Les classes de base héritées virtuellement sont une exception puisqu'elles sont initialisées par leur classe dérivée la plus basse dans l'arbre d'héritage. `Animal` sera donc initialisée par `Pegase` et non par `Cheval` et `Oiseau`. Les constructeurs de ces deux classes doivent initialiser `Animal` dans leurs constructeurs, mais ses initialisations seront ignorées lors de la création d'un objet `Pegase`.

Le Listing 14.6 reprend le code du Listing 14.5, en lui ajoutant des opérations de dérivation virtuelle.

Listing 14.6 : Héritage virtuel

```
0:  // Listing 14.6
1:  // Héritage virtuel
2:  #include <iostream>
3:  using namespace std;
4:
5:  typedef int TAILLE;
6:  enum COULEUR { Rouge, Vert, Bleu, Jaune, Blanc, Noir, Brun } ;
7:
```

```
 8: class Animal        // base commune à Cheval et Oiseau
 9: {
10: public:
11:    Animal(int);
12:    virtual ~Animal() { cout << "Destructeur de Animal...\n"; }
13:    virtual int GetAge() const { return sonAge; }
14:    virtual void SetAge(int age) { sonAge = age; }
15: private:
16:    int sonAge;
17: };
18:
19: Animal::Animal(int age):
20:   sonAge(age)
21: {
22:    cout << "Constructeur de Animal...\n";
23: }
24:
25: class Cheval : virtual public Animal
26: {
27: public:
28:    Cheval(COULEUR couleur, TAILLE taille, int age);
29:    virtual ~Cheval() { cout << "Destructeur de Cheval...\n"; }
30:    virtual void Hennir()const { cout << "Hiihiii!... "; }
31:    virtual TAILLE GetTaille() const { return saTaille; }
32:    virtual COULEUR GetCouleur() const { return saCouleur; }
33: protected:
34:    TAILLE saTaille;
35:    COULEUR saCouleur;
36: };
37:
38: Cheval::Cheval(COULEUR couleur, TAILLE taille, int age):
39:   Animal(age),
40:   saCouleur(couleur),saTaille(taille)
41: {
42:    cout << "Constructeur de Cheval...\n";
43: }
44:
45: class Oiseau : virtual public Animal
46: {
47: public:
48:    Oiseau(COULEUR couleur, bool migrateur, int age);
49:    virtual ~Oiseau() {cout << "Destructeur de Oiseau...\n";  }
50:    virtual void Gazouiller()const { cout << "Cuicui... ";  }
51:    virtual void Voler()const
52:             { cout << "Je vole ! Je vole ! Je vole ! "; }
53:    virtual COULEUR GetCouleur() const { return saCouleur; }
54:    virtual bool GetMigration() const { return saMigration; }
55: protected:
```

```
 56:     COULEUR saCouleur;
 57:     bool saMigration;
 58:  };
 59:
 60:  Oiseau::Oiseau(COULEUR couleur, bool migrateur, int age):
 61:    Animal(age),
 62:    saCouleur(couleur), saMigration(migrateur)
 63:  {
 64:     cout << "Constructeur de Oiseau...\n";
 65:  }
 66:
 67:  class Pegase : public Cheval, public Oiseau
 68:  {
 69:  public:
 70:     void Gazouiller() const { Hennir(); }
 71:     Pegase(COULEUR, TAILLE, bool, long, int);
 72:     virtual ~Pegase() {cout << "Destructeur de Pegase...\n";}
 73:     virtual long GetNbCroyants() const
 74:           { return  nbCroyants; }
 75:     virtual COULEUR GetCouleur() const
75a:          { return Cheval::saCouleur; }
 76:  private:
 77:     long nbCroyants;
 78:  };
 79:
 80:  Pegase::Pegase(
 81:     COULEUR uneCouleur,
 82:     TAILLE uneTaille,
 83:     bool migrateur,
 84:     long nb,
 85:     int age):
 86:       Cheval(saCouleur, saTaille, age),
 87:       Oiseau(saCouleur, migrateur, age),
 88:       Animal(age*2),
 89:       nbCroyants(nb)
 90:  {
 91:     cout << "Constructeur de Pegase...\n";
 92:  }
 93:
 94:  int main()
 95:  {
 96:     Pegase *pPeg = new Pegase(Rouge, 5, true, 10, 2);
 97:     int age = pPeg->GetAge();
 98:     cout << "Pegase a " << age << " ans.\n";
 99:     delete pPeg;
100:    return 0;
101:  }
```

Ce programme produit le résultat suivant :

```
Constructeur de Animal...
Constructeur de Cheval...
Constructeur de Oiseau...
Constructeur de Pegase...
Pegase a 4 ans
Destructeur de Pegase...
Destructeur de Oiseau...
Destructeur de Cheval...
Destructeur de Animal...
```

Aux lignes 25 et 45, `Cheval` et `Oiseau` héritent *virtuellement* de `Animal`. Vous remarquerez que les constructeurs de `Oiseau` et `Cheval` initialisent quand même l'objet `Animal`.

`Pegase` hérite à la fois de `Oiseau` et `Cheval` et, en tant que classe la plus dérivée de `Animal`, initialise également `Animal`. C'est l'initialisation de `Pegase` qui est appelée et les appels au constructeur de `Animal` dans `Oiseau` et `Cheval` sont ignorés. Vous pouvez le vérifier car l'âge de l'objet `Pegase` est passé trois fois au constructeur de `Animal` : par le constructeur de `Pegase` et par les constructeurs de `Cheval` et `Oiseau`. La différence est que `Pegase` double cette valeur avant de la transmettre et c'est bien le double de l'âge (4) qui s'affiche à la ligne 98.

`Pegase` n'a plus besoin de supprimer l'ambiguïté de l'appel à la fonction `GetAge()`, ce qui lui permet d'hériter simplement cette méthode de la classe `Animal`. En revanche, l'appel à la fonction `GetCouleur()` doit être explicite, car celle-ci se trouve dans les deux classes de base, mais pas dans `Animal`.

Déclaration de classes virtuelles

Pour que les classes dérivées ne contiennent qu'une seule instance des classes de base communes, déclarez les classes intermédiaires de sorte qu'elles héritent virtuellement de la classe de base.

Exemple 1

```
class Cheval : virtual public Animal
class Oiseau : virtual public Animal
class Pegase : public Cheval, public Oiseau
```

Exemple 2

```
class Anesse : virtual public Equide
class Cheval : virtual public Equide
class Bardot : public Anesse, public Cheval
```

Problèmes liés à l'héritage multiple

Bien que l'héritage multiple offre plusieurs avantages par rapport à l'héritage simple, nombreux sont les programmeurs qui hésitent à l'adopter dans leurs applications. Leurs objections les plus fréquents sont une difficulté accrue pour le débogage, l'évolution plus complexe et plus risquée des hiérarchies de classes et le fait que quasiment tout ce qui peut être réalisé par l'héritage multiple peut également l'être sans lui. Certains langages, comme Java et C#, n'autorisent d'ailleurs pas l'héritage multiple pour certaines de ces raisons.

Ces objections sont recevables et il est souhaitable d'éviter d'ajouter une complexité inutile dans ses programmes. Certains débogueurs ont du mal avec l'héritage multiple et certaines conceptions sont inutilement obscurcies par une utilisation irréfléchie de l'héritage multiple.

Faire	*Ne pas faire*
● Utiliser l'héritage multiple lorsqu'une nouvelle classe doit faire appel à des fonctions et à des données de plusieurs classes de base. ● Utiliser l'héritage virtuel lorsque la classe dont le niveau de dérivation est le plus bas dans la hiérarchie ne doit avoir qu'une seule instance de la classe de base partagée. ● Initialiser la classe de base partagée depuis la classe dont le niveau de dérivation est le plus bas dans la hiérarchie lorsque vous utilisez des classes de bases virtuelles.	● Utiliser l'héritage multiple lorsqu'un héritage simple suffit.

Mixins et classes de fonctionnalités

Les classes de fonctionnalités (ou classes *mixin*) sont une solution intermédiaire entre l'héritage simple et l'héritage multiple. La classe `Cheval` pourrait, par exemple, dériver de `Animal` et de `Affichable`, cette dernière n'ajoutant que quelques méthodes d'affichage d'objets quelconques.

Un *mixin* est une classe permettant d'ajouter des fonctionnalités spécialisées à une classe dérivée sans introduire beaucoup de méthodes ou de données supplémentaires.

Une classe de fonctionnalités est intégrée à une classe dérivée comme n'importe quelle autre classe, en déclarant que la classe dérivée en hérite publiquement. La seule différence est qu'une classe de fonctionnalités contient peu de données, voire aucune. Il s'agit bien

sûr d'une distinction arbitraire et un mixin n'est finalement qu'une façon simple d'indiquer que, parfois, on souhaite ajouter quelques fonctionnalités supplémentaires à une classe dérivée sans la compliquer outre mesure.

Certains débogueurs traiteront plus aisément les classes dérivées contenant des classes de fonctionnalités que celles qui utilisent l'héritage multiple. En outre, il y a moins de risque d'ambiguïté lors de l'accès aux données de l'autre classe de base principale.

Par exemple, si `Cheval` dérive de `Animal` et de `Affichable`, cette dernière ne contiendra pas de données. `Animal` sera ce qu'elle a toujours été. Toutes les données de `Cheval` proviendront donc de `Animal` alors que ses méthodes seront issues à la fois de `Animal` et de `Affichable`.



Le terme mixin vient d'un glacier de Sommerville dans le Massachusetts, qui mélangeait des bonbons et des biscuits à ses préparations de glaces. Cela a semblé être une bonne métaphore pour certains programmeurs qui avaient l'habitude d'y faire une pause l'été, notamment lorsqu'ils développaient avec le langage de programmation orienté objet SCOOPS.

Types abstraits de données (TAD)

Vous aurez souvent besoin de créer une hiérarchie pour regrouper vos classes. . Vous pourriez ainsi créer une classe `Forme` et en dériver les classes `Rectangle` et `Cercle`. La classe `Carre` peut à son tour dériver de `Rectangle` puisqu'un carré est un rectangle particulier.

Chaque classe dérivée redéfinira les méthodes `Tracer()`, `GetSurface()`, etc. Le Listing 14.7 présente une implémentation dépouillée de la classe `Forme` et de ses classes dérivées `Cercle` et `Rectangle`.

Listing 14.7 : Classes *Forme*

```
 0:  //Listing 14.7. Classes Forme.
 1:
 2:  #include <iostream>
 3:  using std::cout;
 4:  using std::cin;
 5:  using std::endl;
 6:
 7:  class Forme
 8:  {
 9:  public:
10:     Forme(){}
11:     virtual ~Forme(){}
```

```
12:     virtual long GetSurface() { return -1; } // erreur
13:     virtual long GetPerim() { return -1; }
14:     virtual void Tracer() {}
15:  private:
16:  };
17:
18:  class Cercle : public Forme
19:  {
20:  public:
21:     Cercle(int rayon): sonRayon(rayon){}
22:     ~Cercle(){}
23:     long GetSurface() { return 3 * sonRayon * sonRayon; }
24:     long GetPerim() { return 6 * sonRayon; }
25:     void Tracer();
26:  private:
27:     int sonRayon;
28:     int saCirconference;
29:  };
30:
31:  void Cercle::Tracer()
32:  {
33:     cout << "Tracé d'un cercle!\n";
34:  }
35:
36:
37:  class Rectangle : public Forme
38:  {
39:  public:
40:     Rectangle(int longueur, int largeur):
41:       saLongueur(longueur), saLargeur(largeur){}
42:     virtual ~Rectangle(){}
43:     virtual long GetSurface() {return saLongueur * saLargeur;}
44:     virtual long GetPerim()
44a:             { return 2 * saLongueur + 2 * saLargeur; }
45:     virtual int GetLongueur() { return saLongueur; }
46:     virtual int GetLargeur() { return saLargeur; }
47:     virtual void Tracer();
48:  private:
49:     int saLargeur;
50:     int saLongueur;
51:  };
52:
53:  void Rectangle::Tracer()
54:  {
55:     for (int i = 0; i < saLongueur; i++)
56:     {
57:        for (int j = 0; j < saLargeur; j++)
```

```
58:           cout << "x ";
59:
60:     cout << "\n";
61:     }
62:  }
63:
64:  class Carre : public Rectangle
65:  {
66:  public:
67:     Carre(int longueur);
68:     Carre(int longueur, int largeur);
69:     ~Carre(){}
70:     long GetPerim() {return 4 * GetLongueur();}
71:  };
72:
73:  Carre::Carre(int longueur):
74:    Rectangle(longueur, longueur)
75:  {}
76:
77:  Carre::Carre(int longueur, int largeur):
78:    Rectangle(longueur, largeur)
79:  {
80:     if (GetLongueur() != GetLargeur())
81:     cout << "Erreur, ce n'est pas un carré... Rectangle ??\n";
82:  }
83:
84:  int main()
85:  {
86:     int choix;
87:     bool fQuit = false;
88:     Forme * pforme;
89:
90:     while ( !fQuit )
91:     {
92:       cout << "(1)Cercle (2)Rectangle (3)Carré (0)Quitter : ";
93:       cin >> choix;
94:
95:       switch (choix)
96:       {
97:       case 0:   fQuit = true;
98:             break;
99:       case 1: pforme = new Cercle(5);
100:            break;
101:      case 2: pforme = new Rectangle(4,6);
102:            break;
103:      case 3: pforme = new Carre(5);
104:            break;
```

```
105:       default:
106:             cout << "Entrez un nombre entre 0 et 3\n" ;
107:             continue;
108:             break;
109:        }
110:        if( !fQuit )
111:           pforme ->Tracer();
112:        delete pforme
113:        pforme = 0;
114:        cout << endl;
115:     }
116:     return 0;
117:  }
```

Ce programme produit le résultat suivant :

```
(1)Cercle (2)Rectangle (3)Carré (0)Quitter : 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x

(1)Cercle (2)Rectangle (3)Carré (0)Quitter :3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x

(1)Cercle (2)Rectangle (3)Carré (0)Quitter :0
```

La classe `Forme` est déclarée de la ligne 7 à la ligne 16. Ses méthodes `GetSurface()` et `GetPerim()` renvoient une valeur d'erreur, alors que sa méthode `Tracer()` ne fait rien puisqu'on aurait bien du mal à expliquer comment tracer une forme : on ne sait tracer que certains types de formes (les cercles, les rectangles, etc.) ; une forme est une abstraction qui ne peut pas être dessinée.

`Cercle` dérive de `Forme` aux lignes 18 à 29 et redéfinit les trois méthodes virtuelles. Vous remarquerez qu'il n'y aucune raison de répéter le mot-clé `virtual` puisqu'il fait partie de leur héritage, mais cela ne pose aucun problème de le réitérer, comme dans la classe `Rectangle` (lignes 43, 44 et 47). C'est même une pratique conseillée, car c'est une forme de documentation.

`Carre` dérive de `Rectangle` (lignes 64 à 71), redéfinit également la méthode `GetPerim()` et hérite des autres méthodes de la classe `Rectangle`.

Il est cependant troublant qu'un client puisse créer une instance de `Forme` et il serait souhaitable de l'en empêcher. Cette classe n'existe en effet que pour fournir une interface commune aux classes qui en dérivent ; c'est donc un type abstrait de données ou TAD.

Dans une classe *abstraite*, l'interface représente un concept (comme une forme) et non un objet spécifique (comme un cercle). En C++, une classe abstraite est toujours une classe de base pour d'autre classe qu'il n'est pas possible d'instancier.

Fonctions virtuelles pures

C++ gère la création de types abstraits (ou ADT) avec des *fonctions virtuelles pures*. Une fonction virtuelle est dite pure lorsqu'elle est initialisée à zéro :

```
virtual void Tracer() = 0;
```

Dans cet exemple, la classe possède une fonction `Tracer()`, mais une implémentation nulle et elle ne peut pas être appelée. Elle peut toutefois être écrasée dans les classes descendantes.

Toute classe comprenant une ou plusieurs fonctions virtuelles pures est une classe abstraite, qu'il est impossible d'instancier. En fait, il n'est pas permis d'instancier une classe abstraite, ni une classe qui dérive d'une classe abstraite et qui n'implémente pas toutes ses fonctions virtuelles pures. L'ajout d'une fonction virtuelle pure à une classe indique au client que :

- Il est impossible d'en créer un objet. Cette classe doit être dérivée.
- Toutes les fonctions virtuelles pures héritées par une classe doivent être redéfinies pour que l'on puisse en créer des objets.

Toute classe dérivée qui dérive d'une classe abstraite hérite de la fonction virtuelle pure et doit redéfinir cette dernière pour qu'elle puisse devenir elle-même instanciable. Si `Rectangle` hérite de `Forme` et que cette dernière contient trois méthodes virtuelles pures, `Rectangle` doit redéfinir ces trois méthodes sous peine d'être elle-même une classe virtuelle. Le Listing 14.8 réécrit la classe `Rectangle` pour qu'elle soit un type abstrait de données. Pour des raisons de mise en page, le reste du Listing 14.7 n'a pas été inséré ici. Si vous souhaitez exécuter le programme, remplacez la déclaration de `Forme` aux lignes 7 à 17 du Listing 14.7 par la déclaration du Listing 14.8 et recompilez le programme.

Listing 14.8 : Classe abstraite

```
0:  //Listing 14.8 Classes abstraites
1:
2:  class Forme
3:  {
```

```
 4:  public:
 5:        Forme(){}
 6:        ~Forme(){}
 7:        virtual long GetSurface() = 0;
 8:        virtual long GetPerim()= 0;
 9:        virtual void Tracer() = 0;
10:  private:
11:  };
```

Ce programme produit le résultat suivant :

```
(1)Cercle (2)Rectangle (3)Carré (0)Quitter  :2
x x x x x x
x x x x x x
x x x x x x
x x x x x x

(1)Cercle (2)Rectangle (3)Carré (0)Quitter :3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x

(1)Cercle (2)Rectangle (3)Carré (0)Quitter :0
```

Comme vous pouvez le constater, le programme fonctionne correctement. La seule différence est qu'il est désormais impossible de créer des objets de la classe `Forme`.

Types Abstraits de Données (TAD)

Un type abstrait se caractérise par la présence de une ou plusieurs méthodes virtuelles pures dans la déclaration de classe (on dit également que la classe est abstraite). Une méthode virtuelle pure est initialisée à 0.

Exemple

```
class Forme
{
   virtual void Tracer() = 0;    // virtuelle pure
};
```

Implémentation des méthodes virtuelles pures

En général, les méthodes virtuelles pures d'une classe abstraite ne sont jamais implémentées. En effet, comme aucun objet de cette classe ne sera jamais créé, il n'y a aucune raison de fournir des implémentations et la classe abstraite sert uniquement à définir l'interface des objets qui en dérivent.

Il est toutefois possible d'implémenter une méthode virtuelle pure afin qu'elle puisse être appelée par les objets des classes dérivées, par exemple pour fournir une fonctionnalité commune à toutes les redéfinitions de cette méthode. Le Listing 14.9 reprend le Listing 14.7 mais, cette fois-ci, `Forme` est une classe abstraite fournissant une implémentation pour sa méthode virtuelle pure `Tracer()`. La classe `Cercle` redéfinit `Tracer()`, ce qu'elle est obligée de faire pour ne pas être elle-même abstraite, mais réalise un chaînage ascendant vers la méthode de la classe de base.

Listing 14.9 : Implémentation des méthodes virtuelles pures

```
 0:  //Listing 14.9 Implémentation des méthodes virtuelles pures
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Forme
 6:  {
 7:  public:
 8:     Forme(){}
 9:     virtual ~Forme(){}
10:     virtual long GetSurface() = 0;
11:     virtual long GetPerim()= 0;
12:     virtual void Tracer() = 0;
13:  private:
14:  };
15:
16:  void Forme::Tracer()
17:  {
18:     cout << "Mécanisme de dessin abstrait !\n";
19:  }
20:
21:  class Cercle : public Forme
22:  {
23:  public:
24:     Cercle(int rayon): sonRayon(rayon){}
25:     virtual ~Cercle(){}
26:     long GetSurface() { return 3.14 * sonRayon * sonRayon; }
27:     long GetPerim() { return 2 * 3.14 * sonRayon; }
28:     void Tracer();
```

```
29:  private:
30:     int sonRayon;
31:     int saCirconference;
32:  };
33:
34:  void Cercle::Tracer()
35:  {
36:     cout << "Tracé du cercle !\n";
37:     Forme::Tracer();
38:  }
39:
40:
41:  class Rectangle : public Forme
42:  {
43:  public:
44:     Rectangle(int longueur, int largueur):
45:       saLongueur(longueur), saLargeur(largeur){}
46:     virtual ~Rectangle(){}
47:     long GetSurface() { return saLongueur * saLargeur; }
48:     long GetPerim() {return 2 * saLongueur + 2 * saLargeur; }
49:     virtual int GetLongueur() { return saLongueur; }
50:     virtual int GetLargeur() { return saLargeur; }
51:     void Tracer();
52:  private:
53:     int saLargeur;
54:     int saLongueur;
55:  };
56:
57:  void Rectangle::Tracer()
58:  {
59:     for (int i = 0; i < saLongueur; i++)
60:     {
61:        for (int j = 0; j < saLargeur; j++)
62:           cout << "x ";
63:
64:     cout << "\n";
65:     }
66:     Forme::Tracer();
67:  }
68:
69:
70:  class Carre : public Rectangle
71:  {
72:  public:
73:     Carre(int longueur);
74:     Carre(int longueur, int largeur);
75:     virtual ~Carre(){}
```

```
76:     long GetPerim() {return 4 * GetLongueur();}
77:  };
78:
79:  Carre::Carre(int longueur):
80:    Rectangle(longueur, longueur)
81:  {}
82:
83:  Carre::Carre(int longueur, int largeur):
84:    Rectangle(longueur, largeur)
85:
86:  {
87:     if (GetLongueur() != GetLargeur())
88:     cout << "Erreur, ce n'est pas un carré... Rectangle ??\n";
89:  }
90:
91:  int main()
92:  {
93:     int choix;
94:     bool fQuit = false;
95:     Forme * pforme;
96:
97:     while (fQuit == false)
98:     {
99:      cout << "(1)Cercle (2)Rectangle (3)Carré (0)Quitter : ";
100:     cin >> choix;
101:
102:     switch (choix)
103:     {
104:      case 1: pforme = new Cercle(5);
105:              break;
106:      case 2: pforme = new Rectangle(4,6);
107:              break;
108:      case 3: pforme = new Carre (5);
109:              break;
110:      default: fQuit = true;
111:              break;
112:     }
113:     if (fQuit == false)
114:      {
115:      pforme->Tracer();
116:      delete pforme;
117:      cout << endl;
118:      }
119:    }
120:    return 0;
121: }
```

Ce programme produit le résultat suivant :

```
(1)Cercle (2)Rectangle (3)Carré (0)Quitter : 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x
Mécanisme de dessin abstrait !

(1)Cercle (2)Rectangle (3)Carré (0)Quitter : 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x
Mécanisme de dessin abstrait !

(1)Cercle (2)Rectangle (3)Carré (0)Quitter : 0
```

La classe abstraite `Forme` est déclarée de la ligne 5 à la ligne 14 et ses trois méthodes d'accès sont des méthodes virtuelles pures. Ce n'était pas nécessaire, mais c'est une pratique conseillée. Il suffit que l'une des méthodes soit virtuelle pure pour que la classe soit une classe abstraite.

À la différence de `Tracer()` (implémentée lignes 16 à 19), les méthodes `GetSurface()` et `GetPerim()` ne sont pas implémentées. `Cercle` et `Rectangle` redéfinient `Tracer()` et réalisent un chaînage ascendant vers la méthode de la classe de base afin de profiter de la fonctionnalité partagée qu'elle propose.

Hiérarchies complexes d'abstraction

Il est parfois nécessaire de dériver des classes abstraites à partir d'autres classes abstraites. Vous pouvez, par exemple, avoir besoin d'implémenter certaines méthodes virtuelles pures tout en laissant d'autres non pures.

Supposons que vous ayez créé la classe `Animal`, en déclarant les méthodes `Manger()`, `Dormir()`, `Bouger()` et `Reproduire()` comme virtuelles pures. À partir de `Animal`, vous pourrez dériver `Mammifere` et `Poisson`.

Comme vous savez que tous les mammifères se reproduisent de la même façon, vous redéfinissez `Reproduire()` pour qu'elle soit non pure, sans toucher aux autres.

`Chien` dérive de `Mammifere`. Cette classe doit redéfinir et implémenter les trois méthodes virtuelles pures de sa classe de base si l'on veut pouvoir créer des objets `Chien`.

En tant que concepteur, vous indiquez donc que l'on ne pourra pas créer d'objets `Animal` ou `Mammifere`, mais que tous les mammifères pourront hériter de la méthode `Reproduire()` fournie par cette classe sans la redéfinir.

Le Listing 14.10 illustre cette technique d'implémentation d'un TAD.

Listing 14.10 : Dérivation de classes abstraites

```
 0:  // Listing 14.10
 1:  // Dérivation de classes abstraites
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  enum COULEUR { Rouge, Vert, Bleu, Jaune, Blanc, Noir, Brun } ;
 6:
 7:  class Animal     // base commune à Mammiferes et Poisson
 8:  {
 9:  public:
10:     Animal(int);
11:     virtual ~Animal() {cout << "Destructeur de Animal...\n"; }
12:     virtual int GetAge() const { return sonAge; }
13:     virtual void SetAge(int age) { sonAge = age; }
14:     virtual void Dormir() const = 0;
15:     virtual void Manger() const = 0;
16:     virtual void Reproduire() const = 0;
17:     virtual void Bouger() const = 0;
18:     virtual void Crier() const = 0;
19:  private:
20:     int sonAge;
21:  };
22:
23:  Animal::Animal(int age):
24:    sonAge(age)
25:  {
26:     cout << "Constructeur de Animal...\n";
27:  }
28:
29:  class Mammifere : public Animal
30:  {
31:  public:
32:     Mammifere(int age): Animal(age)
33:        { cout << "Constructeur de Mammifere...\n";}
34:     virtual ~Mammifere()
34a:       { cout << "Destructeur de Mammifere...\n";}
35:     virtual void Reproduire() const
36:        { cout << "Reproduction des mammifères...\n"; }
37:  };
```

```
 38:
 39:  class Poisson : public Animal
 40:  {
 41:  public:
 42:     Poisson(int age):Animal(age)
 43:        { cout << "Constructeur de Poisson...\n";}
 44:     virtual ~Poisson()
44a:       {cout << "Destructeur de Poisson...\n";  }
 45:     virtual void Dormir() const
45a:       { cout << "Je fais la sieste...\n"; }
 46:     virtual void Manger() const
46a:       { cout << "Je déjeune...\n"; }
 47:     virtual void Reproduire() const
 48:        { cout << "Je ponds mes oeufs...\n"; }
 49:     virtual void Bouger() const
 50:        { cout << "Je nage...\n";   }
 51:     virtual void Crier() const { }
 52:  };
 53:
 54:  class Cheval : public Mammifere
 55:  {
 56:  public:
 57:     Cheval(int age, COULEUR couleur ):
 58:       Mammifere(age), saCouleur(couleur)
 59:        { cout << "Constructeur de Cheval...\n"; }
 60:     virtual ~Cheval() { cout << "Destructeur de Cheval...\n"; }
 61:     virtual void Crier()const { cout << "Hiihiii!... \n"; }
 62:     virtual COULEUR GetCouleur() const { return saCouleur; }
 63:     virtual void Dormir() const
 64:        { cout << "Je fais la sieste...\n"; }
 65:     virtual void Manger() const {cout << "Mon picotin...\n"; }
 66:     virtual void Bouger() const {cout << "Je galope...\n";}
 67:
 68:  protected:
 69:     COULEUR saCouleur;
 70:  };
 71:
 72:  class Chien : public Mammifere
 73:  {
 74:  public:
 75:     Chien(int age, COULEUR couleur ):
 76:       Mammifere(age), saCouleur(couleur)
 77:        { cout << "Constructeur de Chien...\n"; }
 78:     virtual ~Chien() { cout << "Destructeur de Chien...\n"; }
 79:     virtual void Crier()const { cout << "Ouah Ouah!... \n"; }
 80:     virtual void Dormir() const
80a:       { cout << "Je fais la sieste...\n"; }
```

```
 81:     virtual void Manger() const
81a:       { cout << "Ma pâtée préférée...\n"; }
 82:     virtual void Bouger() const
82a:       { cout << "Une petite course...\n"; }
 83:     virtual void Reproduire() const
 84:        { cout << "J'assure ma descendance...\n"; }
 85:
 86:  protected:
 87:     COULEUR saCouleur;
 88:  };
 89:
 90:  int main()
 91:  {
 92:     Animal *pAnimal=0;
 93:     int choix;
 94:     bool fQuit = false;
 95:
 96:     while (fQuit == false)
 97:     {
 98:        cout <<
98a:       "(1)Chien (2)Cheval (3)Poisson (0)Quitter : ";
 99:        cin >> choix;
100:
101:       switch (choix)
102:       {
103:        case 1: pAnimal = new Chien(5, Brun);
104:              break;
105:        case 2: pAnimal = new Cheval(4, Noir);
106:              break;
107:        case 3: pAnimal = new Poisson (5);
108:              break;
109:        default: fQuit = true;
110:              break;
111:        }
112:        if (fQuit == false)
113:        {
114:        pAnimal->Crier();
115:        pAnimal->Manger();
116:        pAnimal->Reproduire();
117:        pAnimal->Bouger();
118:        pAnimal->Dormir();
119:        delete pAnimal;
120:        cout << endl;
121:      }
122:    }
123:     return 0;
124:  }
```

Ce programme produit le résultat suivant :

```
(1)Chien (2)Cheval (3)Poisson (0)Quitter : 1
Constructeur de Animal...
Constructeur de Mammifere...
Constructeur de Chien...
Ouah Ouah!...
Ma pâtée préférée...
J'assure ma descendance...
Une petite course...
Je fais la sieste...
Destructeur de Chien...
Destructeur de Mammifere...
Destructeur de Animal...

(1)Chien (2)Cheval (3)Poisson (0)Quitter : 0
```

La classe abstraite `Animal` est déclarée de la ligne 7 à la ligne 21. Ses méthodes d'accès à `sonAge` ne sont pas pures et sont partagées par tous les animaux. Cette classe contient cinq méthodes virtuelles pures : `Dormir()`, `Manger()`, `Reproduire()`, `Bouger()` et `Crier()`.

La classe `Mammifere` dérive de `Animal` aux lignes 30 à 38 et n'ajoute aucune données. Elle redéfinit cependant la fonction `Reproduire()` afin de fournir une forme de reproduction commune à tous les mammifères. `Poisson` doit également redéfinir cette méthode, car elle hérite directement de `Animal` et ne peut donc pas utiliser la reproduction des mammifères (et c'est une bonne chose !). Cette redéfinition à lieu aux lignes 47 et 48.

Les classes héritant de `Mammifere` n'ont donc plus besoin de redéfinir la fonction `Reproduire()`, même si elles sont libres de le faire (c'est d'ailleurs ce que fait la classe `Chien`, à la ligne 83). `Poisson`, `Cheval` et `Chien` redéfinissent les méthodes virtuelles pures restantes, afin que l'on puisse créer des objets de ces types.

Dans le corps du programme principal, on utilise un pointeur sur `Animal` afin de traiter un à un les différents objets dérivés. Les méthodes virtuelles sont appelées et, selon les liaisons dynamiques de ce pointeur, c'est la méthode de la classe dérivée appropriée qui sera invoquée.

Tenter d'instancier un objet `Animal` ou un objet `Mammifere` produira une erreur de compilation puisque les deux classes sont abstraites.

Utilité des classes abstraites

Dans un programme, la classe `Animal` sera abstraite alors qu'elle ne le sera pas dans un autre. Quel est le facteur déterminant pour définir une classe abstraite ?

La réponse à cette question dépend non pas d'un facteur intrinsèque du monde réel, mais de son rôle dans votre programme. Si vous définissez la classe `Animal` pour un programme censé représenter une ferme ou un zoo, cette classe pourrait être abstraite, alors que `Chien` serait une classe concrète vous permettant de créer des chiens.

Si, par contre, vous créez une simulation de chenil, la classe `Chien` devrait rester abstraite, alors que les autres classes (`Terrier`, `Levrier`, etc.) devraient pouvoir être instanciées. Le niveau d'abstraction dépend donc de la finesse dont vous avez besoin pour pouvoir distinguer les types.

Faire	*Ne pas faire*
● Utiliser les classes abstraites pour fournir une description commune des fonctionnalités d'un certain nombre de classes dérivées. ● Rendre virtuelle pure toute méthode devant être redéfinie.	● Vouloir créer une instance d'une classe abstraite.

Questions-réponses

Q Que signifie v-ptr ?

R Le v-ptr, ou pointeur de méthode virtuelle, est un détail d'implémentation des méthodes virtuelles. Chaque objet d'une classe ayant des méthodes virtuelles possède un v-ptr, qui pointe vers la table de méthodes virtuelles de cette classe. La table des méthodes virtuelles est consultée dès que le compilateur doit déterminer la méthode à appeler dans une situation particulière.

Q Le filtrage ascendant est-il souhaitable ?

R Oui, si la fonctionnalité est partagée vers le haut. Non, si vous ne faites que déplacer l'interface. Si toutes les classes dérivées ne peuvent pas utiliser une méthode, c'est une erreur de la faire remonter dans une classe de base commune. En effet, cela vous oblige à tester le type de l'objet au moment de l'exécution afin de savoir si vous pouvez appeler cette méthode.

Q Pourquoi faut-il éviter les décisions en fonction du type au moment de l'exécution ?

R Parce que cela indique que la hiérarchie d'héritage pour la classe n'a pas été correctement construite. Il vaut mieux revenir sur ses pas et corriger la conception plutôt qu'essayer de contourner le problème.

Q Pourquoi le transtypage est-il une mauvaise pratique ?

R Le transtypage est acceptable s'il est fait correctement. Cependant, il peut servir à contourner le typage fort de C++, ce que l'on souhaite généralement éviter. Si vous devez tester un type au cours de l'exécution d'un programme, puis transtyper un pointeur, c'est généralement un signe que la conception a été mal faite.

En outre, les méthodes doivent opérer avec le type déclaré de leurs paramètres et des variables membres, et ne pas dépendre de conjectures sur ce que fournira le programme appelant, en vertu d'une espèce de contrat implicite. Si cette hypothèse est erronée, les résultats peuvent être imprévisibles.

Q Pourquoi ne pas rendre toutes les méthodes virtuelles ?

R Les méthodes virtuelles sont gérées au moyen d'une table, ce qui influe sur la taille du programme et ses performances. Si votre programme doit exploiter des classes simples sans classes dérivées, il est inutile de définir des méthodes virtuelles. Toutefois, si cette hypothèse change, vous devrez revenir sur vos pas et rendre virtuelles les fonctions des classes ancêtres, faute de quoi des problèmes inattendus peuvent survenir.

Q Quand faut-il qu'un destructeur soit virtuel ?

R À chaque fois que vous pensez qu'une classe sera dérivée et qu'un pointeur sur la classe de base sera utilisé pour accéder à un objet d'une classe dérivée. La règle générale est que si l'une des méthode de la classe est virtuelle, le destructeur doit l'être aussi.

Q Quel est l'intérêt d'une classe abstraite ? Pourquoi ne pas simplement créer une classe non abstraite et s'interdire de créer des objets de ce type ?

R C++ répertorie les erreurs au cours de la compilation du programme pour vous permettre de les résoudre avant de livrer le programme au client. Si vous créez une classe abstraite, c'est-à-dire une classe comprenant des méthodes virtuelles pures, le compilateur signalera comme erreurs les tentatives de création d'objets de ce type abstrait.

Testez vos connaissances

1. Qu'est-ce qu'un transtypage descendant ?
2. Qu'est-ce qu'un filtrage ascendant ?
3. Si un rectangle arrondi se dessine avec des bords droits et des coins arrondis et que la classe `RectArrondi` hérite à la fois de `Rectangle` et de `Cercle`, qui héritent elles-mêmes de `Forme`, combien de formes sont produites lorsque vous créez un `RectArrondi` ?

4. Si `Cheval` et `Oiseau` héritent de `Animal` selon une relation d'héritage virtuel public, leurs constructeurs initialisent-ils le constructeur `Animal` ? Si `Pegase` hérite de `Cheval` et de `Oiseau`, comment initialise-t-il le constructeur de `Animal` ?
5. Déclarez une classe abstraite `Vehicule`.
6. Supposons qu'une classe de base soit abstraite et contienne trois méthodes virtuelles pures. Combien doivent être redéfinies dans les classes dérivées ?

Exercices

1. Déclarez une classe `Jet` héritant de la classe `Fusee` et de la classe `Avion`.
2. Déclarez une classe `Boeing`, héritant de la classe `Jet` décrite dans l'exercice précédent.
3. Écrivez le code qui dérive `Auto` et `Bus` de la classe `Vehicule`. La classe `Vehicule` doit être abstraite et déclarer deux méthodes virtuelles pures. `Auto` et `Bus` ne doivent pas être des classes abstraites.
4. Modifiez le code de l'Exercice 3 pour que `Auto` soit une classe abstraite et dérivez les classes `Formule1` et `Berline` à partir de `Auto`. Dans la classe `Auto`, fournissez une implémentation de l'une des méthodes virtuelles pures de `Vehicule` et rendez-là non pure.

Partie III

Vous avez terminé la deuxième partie de cet ouvrage. Certains des concepts avancés de la programmation orientée objet, tels que l'encapsulation et le polymorphisme, doivent à présent vous paraître familiers.

Nous allons, dans cette troisième partie, aborder d'abord les fonctions statiques et les fonctions amies avant d'attaquer les concepts avancés de l'héritage. Au Chapitre 17, vous étudierez les flux. Le Chapitre 18 traite en détail des espaces de noms. La notion de modèles est abordée au Chapitre 19. Le Chapitre 20 traite des exceptions et de la gestion d'erreurs. Enfin, le Chapitre 21 examine différents concepts non encore abordés dans ce livre. Suivra une discussion sur les étapes qui vous permettront de devenir un véritable "gourou de la programmation C++".



15

Classes et fonctions spéciales

Au sommaire de ce chapitre

- Partage d'informations entre objets de même type
- Caractéristiques des variables membres et fonctions membres statiques
- Utilisation de ces nouvelles fonctionnalités
- Création et gestion de pointeurs sur les fonctions et les fonctions membres
- Tableaux de pointeurs de fonctions

C++ fournit un certain nombre de fonctionnalités permettant de limiter la portée et l'action des variables et pointeurs. Au cours des chapitres précédents, vous avez appris à créer des variables globales, des variables locales, des pointeurs sur des variables et des variables membres de classe.

Partage d'informations entre objets de même type : données membres statiques

Jusqu'à présent, vous pensiez sans doute que les données d'un objet lui étaient spécifiques et n'étaient pas partagées par les autres objets de la classe. Par exemple, si vous disposez de cinq objets `Chat`, chacun est associé à un âge, un poids, etc. L'âge de l'un n'affecte pas l'âge de l'autre.

Parfois, cependant, il est nécessaire de disposer de données s'appliquant à tous les objets d'un même type. Par exemple, vous pouvez souhaiter connaître le nombre d'objets d'une certaine classe qui ont été créés par le programme, ainsi que le nombre d'entre eux qui existent encore. Les variables membres statiques sont partagées par toutes les instances d'une classe. Elles sont un compromis entre les données globales (visibles par toutes les parties du programme) et les données membres (généralement uniquement accessibles par un objet).

Un membre statique peut être considéré comme un membre appartenant à toute une classe et non à un objet particulier. Il y autant de copies des données membres normales que d'objets, alors qu'il n'y a qu'un seul exemplaire d'un membre statique par classe. Le Listing 15.1 déclare un objet `Chat` avec une donnée membre statique, `NbrChats`, qui permet de mémoriser le nombre de chats qui ont été créés. Pour cela, on incrémente ou on décrémente cette variable statique lorsqu'un objet est créé ou supprimé.

Listing 15.1 : Données membres statiques

```
 0:  //Listing 15.1 Données membres statiques
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Chat
 6:  {
 7:  public:
 8:     Chat(int age): sonAge(age) { NbrChats++; }
 9:     virtual ~Chat() { NbrChats--; }
10:     virtual int GetAge() { return sonAge; }
11:     virtual void SetAge(int age) { sonAge = age; }
12:     static int NbrChats;
13:
14:  private:
15:     int sonAge;
16:  };
17:
18:  int Chat::NbrChats = 0;
19:
```

```
20:  int main()
21:  {
22:     const int MaxChats = 5; int i;
23:     Chat *Panier[MaxChats];
24:
25:     for (i = 0; i < MaxChats; i++)
26:        Panier[i] = new Chat(i);
27:
28:     for (i = 0; i < MaxChats; i++)
29:     {
30:        cout << "Il reste ";
31:        cout << Chat::NbrChats;
32:        cout << " chat(s) !" << endl;
33:        cout << "Suppression de celui qui a ";
34:        cout << Panier[i]->GetAge();
35:        cout << " ans" << endl;
36:        delete Panier[i];
37:        Panier[i] = 0;
38:     }
39:     return 0;
40:  }
```

Ce programme produit le résultat suivant :

```
Il reste 5 chat(s) !
Suppression de celui qui a 0 ans
Il reste 4 chat(s) !
Suppression de celui qui a 1 ans
Il reste 3 chat(s) !
Suppression de celui qui a 2 ans
Il reste 2 chat(s) !
Suppression de celui qui a 3 ans
Il reste 1 chat(s) !
Suppression de celui qui a 4 ans
```

Les lignes 5 à 16 déclare une classe `Chat` simplifiée. La ligne 12 déclare `NbrChats` comme étant une variable membre statique de type entier.

À la différence des variables membres non statiques, la déclaration de `NbrChats` ne définit pas de valeur numérique entière. La création d'un objet `Chat` ne réserve pas de mémoire pour `NbrChats` puisque cette dernière ne fait pas partie de l'objet. Cette variable est définie et initialisée à la ligne 18.

Une erreur classique consiste à oublier de déclarer les variables membres statiques, puis à oublier de les définir. En ce cas, l'éditeur de liens vous en informe à l'aide d'un message du type :

```
undefined symbol Chat::NbrChats
```

Ceci est inutile pour la variable `sonAge` puisqu'il s'agit d'un membre non statique et qu'elle est définie à chaque fois que l'on crée un objet `Chat`, ce que l'on fait à la ligne 26.

Le constructeur de `Chat` incrémente la variable membre statique à la ligne 8. Le destructeur la décrémente à la ligne suivante, ce qui permet de déterminer à tout moment combien d'objets ont été créés, mais pas encore supprimés.

Le programme crée cinq instances de `Chat` aux lignes 20 à 40, puis les copie dans un tableau. Pour cela, il appelle cinq fois le constructeur de `Chat` et la variable statique `NbrChats` est donc incrémentée cinq fois à partir de sa valeur initiale 0.

Le programme parcourt ensuite les cinq positions et affiche la valeur de `NbrChats` avant de supprimer le pointeur sur l'objet `Chat` courant (ligne 36). L'affichage produit montre que l'on part de la valeur 5 (car on a bien créé 5 chats) et qu'il y a un chat de moins à chaque tour de boucle.

Vous remarquerez que `NbrChats` est publique et qu'elle est lue directement depuis la fonction principale du programme. Il n'y aucune raisin de l'exposer ainsi : en fait, il serait préférable de la rendre privée comme les autres membres et d'utiliser une méthode d'accès publique pour la consulter (mais vous devrez alors y accéder *via* une instance de `Chat`). Si vous souhaitez y accéder directement, sans passer par un objet `Chat`, vous avez deux possibilités : conserver la variable publique (comme dans le Listing 15.2) ou créer une méthode membre statique comme nous le verrons plus loin.

Listing 15.2 : Accès aux membres statiques sans passer par un objet

```
 0:  //Listing 15.2 données membres statiques
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Chat
 6:  {
 7:  public:
 8:     Chat(int age): sonAge(age) { NbrChats++; }
 9:     virtual ~Chat() { NbrChats--; }
10:     virtual int GetAge() { return sonAge; }
11:     virtual void SetAge(int age) { sonAge = age; }
12:     static int NbrChats;
13:
14:  private:
15:     int sonAge;
16:  };
17:
18:  int Chat::NbrChats = 0;
19:
```

```
20:  void Telepathie();
21:
22:  int main()
23:  {
24:     const int MaxChats = 5; int i;
25:     Chat *Panier[MaxChats];
26:
27:     for (i = 0; i < MaxChats; i++)
28:     {
29:        Panier[i] = new Chat(i);
30:        Telepathie();
31:     }
32:
33:     for ( i = 0; i < MaxChats; i++)
34:     {
35:        delete Panier[i];
36:        Telepathie();
37:     }
38:     return 0;
39:  }
40:
41:  void Telepathie()
42:  {
43:     cout << "Il reste ";
44:     cout << Chat::NbrChats << " chat(s) !" << endl;
45:  }
```

Ce programme produit le résultat suivant :

```
Il reste 1 chat(s) !
Il reste 2 chat(s) !
Il reste 3 chat(s) !
Il reste 4 chat(s) !
Il reste 5 chat(s) !
Il reste 4 chat(s) !
Il reste 3 chat(s) !
Il reste 2 chat(s) !
Il reste 1 chat(s) !
Il reste 0 chat(s) !
```

Ce listing ressemble beaucoup au précédent, sauf qu'il inclut une nouvelle fonction, `Telepathie()`. Celle-ci, déclarée lignes 41 à 45, ne crée pas d'objet `Chat`, ni n'en prend en paramètre, mais accède pourtant directement à la variable membre `NbrChats`. Cette variable ne fait partie d'aucun objet, mais appartient à la classe où elle est accessible à toute méthode membre. Si elle est publique, cette variable peut également être lue par n'importe quelle fonction du programme, même si celle-ci n'a pas d'instance d'une classe.

L'autre solution consiste à rendre cette variable privée. Vous pourrez alors y accéder par l'intermédiaire d'une méthode membre, mais vous devrez alors passer par un objet de la classe. Le Listing 15.3 montre cette approche, l'autre possibilité – l'utilisation d'une méthode statique – sera étudiée ensuite.

Listing 15.3 : Lecture de membres statiques à l'aide de fonctions membres non statiques

```
 0:  //Listing 15.3 données membres statiques privées
 1:  #include <iostream>
 2:  using std::cout;
 3:  using std::endl;
 4:
 5:  class Chat
 6:  {
 7:  public:
 8:     Chat(int age): sonAge(age) { NbrChats++; }
 9:     virtual ~Chat() { NbrChats--; }
10:     virtual int GetAge() { return sonAge; }
11:     virtual void SetAge(int age) { sonAge = age; }
12:     virtual int GetNbr() { return NbrChats; }
13:
14:  private:
15:     int sonAge;
16:     static int NbrChats;
17:  };
18:
19:  int Chat::NbrChats = 0;
20:
21:  int main()
22:  {
23:     const int MaxChats = 5; int i;
24:     Chat *Panier[MaxChats];
25:
26:     for (i = 0; i < MaxChats; i++)
27:        Panier[i] = new Chat(i);
28:
29:     for (i = 0; i < MaxChats; i++)
30:     {
31:        cout << "Il reste ";
32:        cout << Panier[i]->GetNbr();
33:        cout << " chat(s) !\n";
34:        cout << "Suppression de celui qui a ";
35:        cout << Panier[i]->GetAge() + 2;
```

```
36:         cout << " ans" << endl;
37:         delete Panier[i];
38:         Panier[i] = 0;
39:      }
40:      return 0;
41:  }
```

Ce programme produit le résultat suivant :

```
Il reste 5 chat(s) !
Suppression de celui qui a 2 ans
Il reste 4 chat(s) !
Suppression de celui qui a 3 ans
Il reste 3 chat(s) !
Suppression de celui qui a 4 ans
Il reste 2 chat(s) !
Suppression de celui qui a 5 ans
Il reste 1 chat(s) !
Suppression de celui qui a 6 ans
```

La ligne 16 déclare la variable membre statique `NbrChats` comme privée. Elle n'est donc désormais accessible qu'à partir des fonctions membres.

Bien qu'elle soit statique, la portée de la variable `NbrChats` est encore celle de la classe. Toute fonction membre, comme `GetNbr()`, peut donc y accéder. Cependant, pour qu'une fonction située à l'extérieur de la classe `Chat` puisse appeler `GetNbr()`, elle doit disposer d'un objet `Chat` sur lequel appeler cette méthode.

Faire	*Ne pas faire*
● Utiliser des variables membres statiques pour partager des données entre toutes les instances d'une classe. ● Définir un accès protégé ou privé aux variables membres statiques pour restreindre leur accès.	● Utiliser des variables membres statiques pour stocker les données d'un objet particulier. Les données membres statiques sont partagées par tous les objets de sa classe.

Méthodes membres statiques

Les méthodes membres statiques sont comme les variables membres statiques : elles n'existent pas au niveau d'un objet, mais dans la portée d'une classe. Il est donc possible de les appeler sans avoir d'objet de la classe, comme le montre le Listing 15.4.

Listing 15.4 : Fonctions membres statiques

```
 0:  //Listing 15.4 Fonctions membres statiques
 1:
 2:  #include <iostream>
 3:
 4:  class Chat
 5:  {
 6:  public:
 7:     Chat(int age): sonAge(age) { NbrChats++; }
 8:     virtual ~Chat() { NbrChats--; }
 9:     virtual int GetAge() { return sonAge; }
10:     virtual void SetAge(int age) { sonAge = age; }
11:     static int GetNbr() { return NbrChats; }
12:  private:
13:     int sonAge;
14:      static int NbrChats;
15:  };
16:
17:  int Chat::NbrChats = 0;
18:
19:  void Telepathie();
20:
21:  int main()
22:  {
23:     const int MaxChats = 5;
24:     Chat *Panier[MaxChats]; int i;
25:     for (i = 0; i < MaxChats; i++)
26:     {
27:        Panier[i] = new Chat(i);
28:        Telepathie();
29:     }
30:
31:     for ( i = 0; i < MaxChats; i++)
32:     {
33:        delete Panier[i];
34:        Telepathie();
35:     }
36:     return 0;
37:  }
38:
39:  void Telepathie()
40:  {
41:     std::cout <<"Il reste " << Chat::GetNbr()
42:               <<" chat(s) !" << std::endl;
43:  }
```

Ce programme produit le résultat suivant :

```
Il reste 1 chat(s) !
Il reste 2 chat(s) !
Il reste 3 chat(s) !
Il reste 4 chat(s) !
Il reste 5 chat(s) !
Il reste 4 chat(s) !
Il reste 3 chat(s) !
Il reste 2 chat(s) !
Il reste 1 chat(s) !
Il reste 0 chat(s) !
```

L'accès à la variable membre statique `NbrChats` est privé, comme l'indique la ligne 14 dans la déclaration de `Chat`. La fonction publique d'accès, `GetNbr()`, est déclarée publique et statique à la ligne 11.

`GetNbr()` étant publique, cette fonction est accessible aux autres fonctions et, puisqu'elle est statique, elle n'a pas besoin d'un objet `Chat` sur lequel être appelée. À la ligne 41, la fonction `Telepathie()` peut donc appeler la méthode d'accès statique sans avoir recours à un objet `Chat`. Notez toutefois que la méthode est totalement qualifiée lorsqu'elle est appelée, ce qui signifie que l'appel de la méthode est préfixé avec le nom de classe suivi du signe deux-points :

```
Chat::GetNbr()
```

Bien entendu, vous auriez pu appeler `GetNbr()` sur les objets `Chat` disponibles dans `main()`, comme pour n'importe quelle autre méthode publique d'accès.



Les méthodes membres statiques ne contenant pas de pointeur `this`, elles ne peuvent être déclarées `const`. De même, dans les méthodes, les variables membres sont lues à l'aide du pointeur `this` : cela signifie que les méthodes membres statiques ne peuvent pas accéder aux variables membres non statiques !

Méthodes membres statiques

Pour accéder aux méthodes membres statiques, il suffit de les appeler sur un objet de la classe, comme vous le feriez pour n'importe quelle méthode membre. Vous pouvez également les appeler sans objet, en indiquant clairement le nom de la classe et le nom de l'objet.

Exemple

```
class Chat
{
public:
     static int GetNbr() { return NbrChats; }
private:
     static int NbrChats;
};
   int Chat::NbrChats = 0;
int main()

  int Nbr;
  Chat leChat;              // définir un chat
  Nbr = leChat.GetNbr();   // accès par objet
  Nbr = Chat::GetNbr();    // accès sans objet
```

Pointeurs sur des fonctions

Exactement comme un nom de tableau est un pointeur constant sur le premier élément du tableau, un nom de fonction est un pointeur constant sur la fonction. Il est possible de déclarer une variable pointeur sur une fonction, puis d'appeler la fonction à l'aide de ce pointeur. Cette fonctionnalité est très pratique puisqu'elle permet d'appeler des fonctions selon ce qui a été saisi par l'utilisateur.

La seule difficulté des pointeurs de fonctions est de comprendre le type d'objet pointé. Un pointeur sur un type `int` pointe sur une variable entière, alors qu'un pointeur sur une fonction doit pointer sur une fonction d'un type de résultat et d'une signature appropriés.

Dans la déclaration :

```
long (* ptrFction) (int);
```

`ptrFction` est déclaré comme un pointeur (reconnaissable à l'astérisque devant le nom) sur une fonction prenant un paramètre entier et renvoyant un entier `long`. Les parenthèses encadrant `* ptrFction` sont nécessaires car celles autour de `int` sont plus prioritaires que l'opérateur d'indirection (`*`). Sans ces premières parenthèses, cette ligne déclarerait donc une fonction prenant un paramètre entier et renvoyant un pointeur sur un type `long` (n'oubliez pas que les espaces n'ont aucune importance ici).

Examinez les deux déclarations suivantes :

```
long * Fonction (int);
long (* ptrFonction) (int);
```

Dans la première, `Fonction()` prend un entier et renvoie un pointeur sur une variable de type `long`. Dans la seconde, `prtFonction` est un pointeur sur une fonction prenant un entier en paramètre et renvoyant une variable de type `long`.

Une déclaration de pointeur de fonction doit toujours comprendre le type du résultat et des parenthèses entourant les types des paramètres.

Listing 15.5 : Pointeurs sur fonctions

```
 0:  // Listing 15.5 Utilisation de pointeurs de fonction
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  void Carre (int&,int&);
 6:  void Cube (int&, int&);
 7:  void Permuter (int&, int &);
 8:  void GetVals(int&, int&);
 9:  void AffVals(int, int);
10:
11:  int main()
12:  {
13:     void (* pFonc) (int &, int &);
14:     bool fQuit = false;
15:
16:     int Val1 = 1, Val2 = 2;
17:     int choix;
18:     while (fQuit == false)
19:     {
20:        cout << "(0)Quitter (1)Modifier Valeurs (2)Carré ";
20a:       cout << "(3)Cube (4)Permuter : ";
21:        cin >> choix;
22:        switch (choix)
23:        {
24:        case 1: pFonc = GetVals; break;
25:        case 2: pFonc = Carre; break;
26:        case 3: pFonc = Cube; break;
27:        case 4: pFonc = Permuter; break;
28:        default: fQuit = true; break;
29:        }
30:
31:        if (fQuit == false)
32:        {
33:           AffVals(Val1, Val2);
34:           pFonc(Val1, Val2);
35:           AffVals(Val1, Val2);
36:        }
```

```
37:      }
38:     return 0;
39:  }
40:
41:  void AffVals(int x, int y)
42:  {
43:     cout << "x : " << x << " y : " << y << endl;
44:  }
45:
46:  void Carre(int & rX, int & rY)
47:  {
48:     rX *= rX;
49:     rY *= rY;
50:  }
51:
52:  void Cube(int & rX, int & rY)
53:  {
54:     int tmp;
55:
56:     tmp = rX;
57:     rX *= rX;
58:     rX = rX * tmp;
59:
60:     tmp = rY;
61:     rY *= rY;
62:     rY = rY * tmp;
63:  }
64:
65:  void Permuter(int & rX, int & rY)
66:  {
67:     int temp;
68:     temp = rX;
69:     rX = rY;
70:     rY = temp;
71:  }
72:
73:  void GetVals (int & rVal1, int & rVal2)
74:  {
75:     cout << "Nouvelle valeur de Val1 : ";
76:     cin >> rVal1;
77:     cout << "Nouvelle valeur de Val2 : ";
78:     cin >> rVal2;
79:  }
```

Ce programme produit le résultat suivant :

```
0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 1
x : 1 y : 2
Nouvelle valeur de Val1 : 2
Nouvelle valeur de Val2 : 3
x : 2 y : 3
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 3
x : 2 y : 3
x : 8 y : 27
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 2
x : 8 y : 27
x : 64 y : 729
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 4
x : 64 y : 729
x : 729 y : 64
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 0
```

Les lignes 5 à 8 déclare quatre fonctions renvoyant `void` et prenant en paramètre deux références à des entiers.

La ligne 13 déclare `pFonc` comme un pointeur sur une fonction qui renvoie `void` et prend deux références à des entiers en paramètres. Comme leurs signatures correspondent, il peut donc pointer sur les quatre fonctions. L'affectation de `pFonc` dépend du choix de l'utilisateur. La valeur des deux entiers est affichée (lignes 33 à 35), la fonction choisie est appelée, puis les valeurs s'affichent de nouveau.

Pointeur de fonction

Un pointeur de fonction est appelé de la même façon que les fonctions sur lesquelles il pointe, sauf que c'est le nom du pointeur de fonction qui est utilisé à la place du nom de la fonction.

On affecte un pointeur à une fonction spécifique en lui affectant le nom de la fonction sans parenthèses. Le nom de la fonction est un pointeur constant sur la fonction elle-même. Le pointeur de fonction est utilisé comme un nom de fonction classique. La valeur renvoyée et la signature déclarées pour le pointeur doivent correspondre à celles de la fonction que vous lui affectez.

Exemple

```
long (*pFoncUne) (int, int);
long Fonction (int, int);
pFoncUne = Fonction;
pFoncUne(5,7);
```



Sachez que les pointeurs de fonctions peuvent être très dangereux. Vous pouvez affecter accidentellement une fonction à un pointeur alors que vous vouliez l'appeler ou appeler la fonction par inadvertance alors que vous vouliez l'affecter à son pointeur.

Utilité des pointeurs de fonctions

Il est généralement déconseillé d'utiliser les pointeurs de fonctions. Ils datent du langage C, avant la programmation orientée objet, où on les utilisait pour mettre en œuvre un style de programmation permettant d'obtenir quelques-uns des avantages de l'orientation objet. Si vous écrivez un programme hautement dynamique et que vous devez gérer des fonctionnalités différentes en fonction de décisions prises en cours d'exécution, cette solution peut toutefois être intéressante.

Le programme du Listing 15.5 aurait certainement pu être écrit sans pointeur de fonctions, mais leur utilisation a permis d'expliciter l'intention et l'utilisation du programme : choisir une fonction dans une liste et l'appeler.

Le Listing 15.6 reprend les prototypes et les définitions du Listing 15.5, mais le corps du programme n'utilise plus de pointeur de fonction. Vous pourrez ainsi comparer les deux approches.

Listing 15.6 : Ce programme reprend le listing précédent, sans pointeur de fonction

```
 0:  // Listing 15.6 - Sans pointeur de fonctions
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  void Carre(int&,int&);
 6:  void Cube(int&, int&);
 7:  void Permuter(int&, int &);
 8:  void GetVals(int&, int&);
 9:  void AffVals(int, int);
10:
11:  int main()
12:  {
13:     bool fQuit = false;
14:     int Val1 = 1, Val2 = 2;
15:     int choix;
16:     while (fQuit == false)
17:     {
18:        cout << "(0)Quitter (1)Modifier Valeurs (2)Carré ";
18a:       cout << "(3)Cube (4)Permuter : ";
19:        cin >> choix;
```

```
20:        switch (choix)
21:        {
22:        case 1:
23:           AffVals(Val1, Val2);
24:           GetVals(Val1, Val2);
25:           AffVals(Val1, Val2);
26:           break;
27:
28:        case 2:
29:           AffVals (Val1, Val2);
30:           Carre(Val1,Val2);
31:           AffVals (Val1, Val2);
32:           break;
33:
34:        case 3:
35:           AffVals(Val1, Val2);
36:           Cube(Val1, Val2);
37:           AffVals (Val1, Val2);
38:           break;
39:
40:        case 4:
41:           AffVals (Val1, Val2);
42:           Permuter(Val1, Val2);
43:           AffVals (Val1, Val2);
44:           break;
45:
46:        default :
47:           fQuit = true;
48:           break;
49:        }
50:     }
51:     return 0;
52:  }
53:
54:  void AffVals(int x, int y)
55:  {
56:     cout << "x: " << x << " y: " << y << endl;
57:  }
58:
59:  void Carre(int & rX, int & rY)
60:  {
61:     rX *= rX;
62:     rY *= rY;
63:  }
64:
65:  void Cube(int & rX, int & rY)
66:  {
67:     int tmp;
```

```
68:
69:     tmp = rX;
70:     rX *= rX;
71:     rX = rX * tmp;
72:
73:     tmp = rY;
74:     rY *= rY;
75:     rY = rY * tmp;
76:  }
77:
78:  void Permuter(int & rX, int & rY)
79:  {
80:     int temp;
81:     temp = rX;
82:     rX = rY;
83:     rY = temp;
84:  }
85:
86:  void GetVals (int & rVal1, int & rVal2)
87:  {
88:     cout << "Nouvelle valeur de Val1 : ";
89:     cin >> rVal1;
90:     cout << "Nouvelle valeur de Val2 : ";
91:     cin >> rVal2;
92:  }
```

Ce programme produit le résultat suivant :

```
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 1
x: 1 y: 2
Nouvelle valeur de Val1 : 2
Nouvelle valeur de Val2 : 3

(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 3
x: 2 y: 3
x: 8 y: 27
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 2
x: 8 y: 27
x: 64 y: 729
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 4
x: 64 y: 729
x: 729 y: 64
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 0
```

Il aurait été tentant de placer la fonction `AffVals()` au début et à la fin de la boucle `while`, plutôt que dans chaque instruction `case` mais, dans ce cas, elle aurait été appelée une fois

de trop (au niveau de l'instruction `case` correspondant à Quitter) et cela ne faisait pas partie de la spécification.

Avec les appels répétés et la redondance du code, le programme devient rapidement confus et difficile à gérer. Dans une application professionnelle, l'avantage des pointeurs de fonctions est encore plus clair : ils permettent d'éviter la duplication du code, de le rendre plus clair et de mettre en place des tables de fonctions qui pourront être appelées en fonction de conditions testées au moment de l'exécution.



La programmation orientée objet permet généralement d'éviter la création ou le passage des pointeurs de fonctions en paramètres. Appelez plutôt la méthode souhaitée de l'objet concerné ou invoquez la méthode de classe voulue. Si vous avez besoin d'un tableau de pointeurs de fonctions, demandez-vous si un tableau d'objets ne serait pas plus approprié.

Appels raccourcis

Il est inutile de déréférencer un pointeur de fonction, même si cela est autorisé. Si `pFonc` est un pointeur sur une fonction prenant un entier en paramètre et renvoyant un résultat de type `long`, et que vous lui affectez une fonction qui correspond à ces conditions, vous pouvez appeler cette fonction de la façon suivante :

```
pFonc(x);
```

ou

```
(*pFonc)(x);
```

Les deux expressions ont le même effet, la première est simplement un raccourci de la seconde.

Tableaux de pointeurs de fonctions

Vous pouvez créer un tableau de pointeurs sur des fonctions renvoyant un certain type de résultat et ayant une signature spécifique, exactement comme vous déclarez un tableau de pointeurs sur des entiers . Le Listing 15.7 reprend le Listing 15.5, mais en utilisant ce type de tableau.

Listing 15.7 : Utilisation d'un tableau de pointeurs sur des fonctions

```
0:  // Listing 15.7
1:  // utilisation d'un tableau de pointeurs sur fonctions
2:
```

```
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  void Carre(int&,int&);
 7:  void Cube(int&, int&);
 8:  void Permuter(int&, int &);
 9:  void GetVals(int&, int&);
10: void AffVals(int, int);
11:
12:  int main()
13:  {
14:     int val1 = 1, val2 = 2;
15:     int choix, i;
16:     const Max = 5;
17:     void (*TableauPtrFonctions[Max])(int&, int&);
18:
19:     for (i = 0; i < Max; i++)
20:     {
21:        cout << "(0)Quitter (1)Modifier Valeurs (2)Carré ";
21a:       cout << "(3)Cube (4)Permuter : ";
22:        cin >> choix;
23:        switch (choix)
24:        {
25:        case 1: TableauPtrFonctions[i] = GetVals; break;
26:        case 2: TableauPtrFonctions[i] = Carre; break;
27:        case 3: TableauPtrFonctions[i] = Cube; break;
28:        case 4: TableauPtrFonctions[i] = Permuter; break;
29:        default: TableauPtrFonctions[i] = 0;
30:        }
31:     }
32:
33:     for (i = 0; i < Max; i++)
34:     {
35:        if (TableauPtrFonctions[i] == 0 )
36:           continue;
37:        TableauPtrFonctions[i](val1,val2);
38:        AffVals(val1,val2);
39:     }
40:     return 0;
41:  }
42:
43:  void AffVals(int x, int y)
44:  {
45:     cout << "x: " << x << " y: " << y << endl;
46:  }
47:
48:  void Carre(int & rX, int & rY)
49:  {
```

```
50:     rX *= rX;
51:     rY *= rY;
52:  }
53:
54:  void Cube(int & rX, int & rY)
55:  {
56:     int tmp;
57:
58:     tmp = rX;
59:     rX *= rX;
60:     rX = rX * tmp;
61:
62:     tmp = rY;
63:     rY *= rY;
64:     rY = rY * tmp;
65:  }
66:
67:  void Permuter(int & rX, int & rY)
68:  {
69:     int temp;
70:     temp = rX;
71:     rX = rY;
72:     rY = temp;
73:  }
74:
75:  void GetVals(int & rVal1, int & rVal2)
76:  {
77:     cout << "Nouvelle valeur de Val1 : ";
78:     cin >> rVal1;
79:     cout << "Nouvelle valeur de Val2 : ";
80:     cin >> rVal2;
81:  }
```

Ce programme produit le résultat suivant :

```
(1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 1
(1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 2
(1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 3
(1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 4
(1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 2
Nouvelle valeur de Val1 : 2
Nouvelle valeur de Val2 : 3
x: 2 y: 3
x: 4 y: 9
x: 64 y: 729
x: 729 y: 64
x: 531441 y: 4096
```

La ligne 17 déclare `TableauPtrFonctions` comme un tableau de cinq pointeurs de fonctions renvoyant `void` et prenant deux références entières en paramètres.

De la ligne 19 à la ligne 31, l'utilisateur est invité à choisir une fonction et chaque élément du tableau reçoit l'adresse de la fonction choisie. De la ligne 33 à la ligne 39, les fonctions sont appelées les unes après les autres et le résultat s'affiche après chaque appel.

Passage de pointeurs de fonctions à d'autres fonctions

Les pointeurs de fonctions et les tableaux de pointeurs de fonctions peuvent être passés en paramètres à d'autres fonctions qui pourront ainsi appeler la fonction appropriée en passant par un pointeur.

Il serait par exemple possible d'améliorer le Listing 15.5 en passant le pointeur de la fonction choisie à une autre fonction (en dehors de `main()`) qui affichera les valeurs, appelera la fonction appropriée, puis réaffichera les valeurs. C'est ce que fait le Listing 15.8.

Listing 15.8 : Passage de pointeurs de fonctions en paramètres

```
 0:  // Listing 15.8 - sans pointeurs sur fonction
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  void Carre(int&,int&);
 6:  void Cube(int&, int&);
 7:  void Permuter(int&, int &);
 8:  void GetVals(int&, int&);
 9:  void AffVals(void (*)(int&, int&),int&, int&);
10:
11:  int main()
12:  {
13:     int val1 = 1, val2 = 2;
14:     int choix;
15:     bool fQuit = false;
16:
17:     void (*pFonc)(int&, int&);
18:
19:     while (fQuit == false)
20:     {
21:        cout << "(0)Quitter (1)Modifier Valeurs (2)Carré ";
21a:       cout << "(3)Cube (4)Permuter : ";
22:        cin >> choix;
23:        switch (choix)
24:        {
25:        case 1:   pFonc = GetVals; break;
```

```
26:        case 2:   pFonc = Carre; break;
27:        case 3:   pFonc = Cube; break;
28:        case 4:   pFonc = Permuter; break;
29:        default:  fQuit = true; break;
30:        }
31:
32:        if (fQuit == false)
33:          AffVals( pFonc, val1, val2);
34:     }
35:
36:     return 0;
37:  }
38:
39:  void AffVals( void (*pFonc)(int&, int&),int& x, int& y)
40:  {
41:     cout << "x: " << x << " y: " << y << endl;
42:     pFonc(x,y);
43:     cout << "x: " << x << " y: " << y << endl;
44:  }
45:
46:  void Carre(int & rX, int & rY)
47:  {
48:     rX *= rX;
49:     rY *= rY;
50:  }
51:
52:  void Cube(int & rX, int & rY)
53:  {
54:     int tmp;
55:
56:     tmp = rX;
57:     rX *= rX;
58:     rX = rX * tmp;
59:
60:     tmp = rY;
61:     rY *= rY;
62:     rY = rY * tmp;
63:  }
64:
65:  void Permuter(int & rX, int & rY)
66:  {
67:     int temp;
68:     temp = rX;
69:     rX = rY;
70:     rY = temp;
71:  }
72:
```

```
73:  void GetVals(int & rVal1, int & rVal2)
74:  {
75:     cout << "Nouvelle valeur de Val1 : ";
76:     cin >> rVal1;
77:     cout << "Nouvelle valeur de Val2 : ";
78:     cin >> rVal2;
79:  }
```

Ce programme produit le résultat suivant :

```
(0)Quitter (1)Modifier valeurs (2)Carré (3)Cube (4)Permuter : 1
x: 1 y:2
Nouvelle valeur de Val1 : 2
Nouvelle valeur de Val2 : 3
x: 2 y: 3
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 3
x: 2 y: 3
x: 8 y: 27
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 2
x: 8 y: 27
x: 64 y: 729
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 4
x: 64 y: 729
x:729 y: 64
(0)Quitter (1)Modifier Valeurs (2)Carre (3)Cube (4)Permuter : 0
```

La ligne 17 déclare `pFonc` comme un pointeur sur une fonction renvoyant `void` et prenant deux paramètres (des références à des entiers). La ligne 9 déclare `AffVals()` comme une fonction acceptant trois paramètres : le premier est un pointeur sur une fonction qui renvoie `void` et attend deux paramètres qui sont des références à des entiers ; les deuxième et troisième paramètres de `AffVals()` sont des références entières. Les lignes 19 et 20 demandent à l'utilisateur de choisir une fonction et la ligne 33 appelle `AffVals()` en lui passant le pointeur de fonction `pFonc` comme premier paramètre.

Demandez à un programmeur C++ ce que signifie cette déclaration :

```
void AffVals(void (*)(int&, int&),int&, int&);
```

C'est une déclaration peu commune dont vous vérifierez probablement la syntaxe dans un livre à chaque fois que vous en avez besoin, mais qui sauvera votre programme dans les rares occasions où cette construction est exactement celle qui convient.

Utilisation de *typedef* avec des pointeurs de fonctions

L'expression `void (*)(int&, int&)` est, pour le moins, peu évidente. Il est possible de la simplifier à l'aide de `typedef`, en déclarant un type (qui est appelé `PFV` dans le

Listing 15.9) représentant un pointeur sur une fonction renvoyant `void` et prenant comme paramètres deux références sur des entiers.

Le Listing 15.9 réécrit le Listing 15.8 en utilisant l'instruction `typedef`.

Listing 15.9 : Utilisation de *typedef* pour rendre plus lisibles les pointeurs de fonctions

```
 0:  // Listing 15.9. Utilisation de typedef
 1:  // pour rendre plus lisibles les pointeurs sur fonctions
 2:
 3:  #include <iostream>
 4:  using namespace std;
 5:
 6:  void Carre(int&,int&);
 7:  void Cube(int&, int&);
 8:  void Permuter(int&, int &);
 9:  void GetVals(int&, int&);
10:  typedef  void (*PFV) (int&, int&) ;
11:  void AffVals(PFV, int&, int&);
12:
13:  int main()
14:  {
15:     int val1 = 1, val2 = 2;
16:     int choix;
17:     bool fQuit = false;
18:
19:     PFV pFonc;
20:
21:     while (fQuit == false)
22:     {
23:        cout << "(0)Quitter (1)Modifier Valeurs (2)Carré ";
23a:      cout << "(3)Cube (4)Permuter : ";
24:        cin >> choix;
25:        switch (choix)
26:        {
27:        case 1:   pFonc = GetVals; break;
28:        case 2:   pFonc = Carre; break;
29:        case 3:   pFonc = Cube; break;
30:        case 4:   pFonc = Permuter; break;
31:        default:  fQuit = true; break;
32:        }
33:
34:        if (fQuit == false)
35:          AffVals( pFonc, val1, val2);
36:     }
37:     return 0;
38:  }
```

```
39:
40:  void AffVals(PFV pFonc, int& x, int& y)
41:  {
42:     cout << "x: " << x << " y: " << y << endl;
43:     pFonc(x, y);
44:     cout << "x: " << x << " y: " << y << endl;
45:  }
46:
47:  void Carre(int & rX, int & rY)
48:  {
49:     rX *= rX;
50:     rY *= rY;
51:  }
52:
53:  void Cube(int & rX, int & rY)
54:  {
55:     int tmp;
56:
57:     tmp = rX;
58:     rX *= rX;
59:     rX = rX * tmp;
60:
61:     tmp = rY;
62:     rY *= rY;
63:     rY = rY * tmp;
64:  }
65:
66:  void Permuter(int & rX, int & rY)
67:  {
68:     int temp;
69:     temp = rX;
70:     rX = rY;
71:     rY = temp;
72:  }
73:
74:  void GetVals(int & rVal1, int & rVal2)
75:  {
76:     cout << "Nouvelle valeur de Val1 : ";
77:     cin >> rVal1;
78:     cout << "Nouvelle valeur de Val2 : ";
79:     cin >> rVal2;
80:  }
```

Ce programme produit le résultat suivant :

```
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 1
x: 1 y: 2
```

```
Nouvelle valeur de Val1 : 2
Nouvelle valeur de Val2 : 3
x: 2 y:3
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 3
x: 2 y:3
x: 8 y: 27
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 2
x: 8 y: 27
x: 64 y: 729
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 4
x: 64 y: 729
x: 729 y: 64
(0)Quitter (1)Modifier Valeurs (2)Carré (3)Cube (4)Permuter : 0
```

L'instruction `typedef` de la ligne 10 déclare `PFV` comme un alias du type "pointeur sur une fonction renvoyant `void` et prenant deux paramètres (des références d'entiers)".

À la ligne suivante, la fonction `AffVals()` attend trois paramètres : un `PFV` et deux références à des entiers. La ligne 19 déclare `pFonc` comme étant désormais de type `PFV`.

Grâce à la définition du type `PFV`, les déclarations de `pFonc` et de `AffVals()` sont bien plus claires et, comme vous pouvez le constater, le résultat est identique. N'oubliez pas, cependant, que `typedef` effectue essentiellement un remplacement : il ne "crée" pas un nouveau type. Ici, son utilisation simplifie beaucoup la relecture du code.

Pointeurs sur des fonctions membres

Jusqu'à présent, tous les pointeurs de fonctions étaient destinés à des fonctions standard, non à des méthodes de classe, mais il est évidemment possible de définir des pointeurs sur des fonctions membres. Il s'agit d'une technique très avancée et peu usitée, qui doit être évitée autant que possible, mais il est préférable de la connaître car certains programmeurs l'utilisent.

Pour créer un pointeur vers une méthode membre, on utilise la même syntaxe que pour les pointeurs sur les fonctions standard, mais on y ajoute le nom de la classe et l'opérateur de portée (`::`). Si `pFonc` pointe sur une fonction membre de la classe `Forme`, prenant en paramètre deux entiers et renvoyant `void`, sa déclaration sera donc :

```
void (Forme::*pFonc) (int, int);
```

Les pointeurs vers les méthodes membres sont utilisés exactement comme les pointeurs de fonctions, sauf qu'ils doivent être appelés sur un objet de la classe appropriée. Le Listing 15.10 montre comment faire.

Listing 15.10 : Pointeurs sur fonctions membres

```
 0:  //Listing 15.10 Pointeurs sur fonctions membres
 1:  // utilisant des méthodes virtuelles
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Mammifere
 6:  {
 7:  public:
 8:     Mammifere(): sonAge(1) {  }
 9:     virtual ~Mammifere() { }
10:     virtual void Crier() const = 0;
11:     virtual void Bouger() const = 0;
12:  protected:
13:     int sonAge;
14:  };
15:
16:  class Chien : public Mammifere
17:  {
18:  public:
19:     void Crier()const { cout << "Ouah Ouah !" << endl; }
20:     void Bouger() const { cout << "Au pied..." << endl; }
21:  };
22:
23:
24:  class Chat : public Mammifere
25:  {
26:  public:
27:     void Crier()const { cout << "Miaou !" << endl; }
28:     void Bouger() const { cout << "Nonchalant..." << endl; }
29:  };
30:
31:
32:  class Cheval : public Mammifere
33:  {
34:  public:
35:     void Crier()const { cout << "hihiii !" << endl; }
36:     void Bouger() const { cout << "Au galop..." << endl; }
37:  };
38:
39:
40:  int main()
41:  {
42:     void (Mammifere::*pFonc)() const =0;
43:     Mammifere* ptr =0;
44:     int Animal;
45:     int Methode;
```

```
46:     bool fQuit = false;
47:
48:     while (fQuit == false)
49:     {
50:        cout << "(0)Quitter (1)Chien (2)Chat (3)Cheval : ";
51:        cin >> Animal;
52:        switch (Animal)
53:        {
54:        case 1:   ptr = new Chien; break;
55:        case 2:   ptr = new Chat; break;
56:        case 3:   ptr = new Cheval; break;
57:        default: fQuit = true; break;
58:        }
59:        if (fQuit == false)
60:        {
61:        cout << "(1)Crier (2)Bouger : ";
62:        cin >> Methode;
63:        switch (Methode)
64:        {
65:           case 1: pFonc = Mammifere::Crier; break;
66:           default: pFonc = Mammifere::Bouger; break;
67:        }
68:
69:           (ptr->*pFonc)();
70:           delete ptr;
71:        }
72:     }
73:     return 0;
74:  }
```

Ce programme produit le résultat suivant :

```
(0)Quitter (1)Chien (2)Chat (3)Cheval: 1
(1)Crier (2)Bouger: 1
Ouah Ouah !
(0)Quitter (1)Chien (2) Chat (3)Cheval: 2
(1)Crier (2)Bouger: 1
Miaou !
(0)Quitter (1)Chien (2) Chat (3)Cheval: 3
(1)Crier (2)Bouger : 2
Au galop...
(0)Quitter (1)Chien (2) Chat (3)Cheval: 0
```

La classe abstraite `Mammifere` est déclarée de la ligne 5 à la ligne 14, avec deux fonctions virtuelles pures : `Crier()` et `Bouger()`. Cette classe est héritée par les classes dérivées `Chien`, `Chat` et `Cheval`, qui redéfinissent `Crier()` et `Bouger()`.

La fonction principale commence à la ligne 40. À la ligne 50, l'utilisateur doit choisir entre trois types d'animaux. Selon son choix, une nouvelle classe dérivée (`Animal`) est créée sur le tas et affectée à `ptr` (lignes 54 à 56).

À la ligne 61, l'utilisateur doit choisir la méthode à appeler. La méthode choisie, `Crier()` ou `Bouger()`, est affectée au pointeur `pFonc`, aux lignes 65 et 66. À la ligne 69, elle est appelée par l'objet créé à l'aide de `ptr` pour accéder à cet objet et de `pFonc` pour accéder à la méthode.

Enfin, `delete` supprime le pointeur `ptr` et libère l'espace mémoire occupé sur le tas (ligne 70). Il n'y a pas de raison d'appeler `delete` sur `pFonc` car c'est un pointeur sur du code, pas sur un objet du tas. Si vous tentiez cette opération, le compilateur vous signalerait une erreur.

Tableaux de pointeurs sur des fonctions membres

Comme les pointeurs de fonctions, les pointeurs de fonctions membres peuvent être stockés dans un tableau. Ce dernier peut être initialisé avec les adresses des différentes fonctions membres, qui seront ensuite appelées par leurs indices dans le tableau. Le Listing 15.11 illustre cette technique.

Listing 15.11 : Tableau de pointeurs sur des fonctions membres

```
 0:  //Listing 15.11 Tableau de pointeurs sur des fonctions membres
 1:  #include <iostream>
 2:  using std::cout;
 3:  using std::endl;
 4:
 5:  class Chien
 6:  {
 7:  public:
 8:     void Crier()const { cout << "Ouah Ouah !" << endl; }
 9:     void Bouger() const { cout << "Au pied..." << endl; }
10:     void Manger() const { cout << "Miam..." << endl; }
11:     void Grogner() const { cout << "Grrrrr" << endl; }
12:     void Couiner() const { cout << "Couine..." << endl; }
13:     void Jouer() const { cout << "Joue..." << endl; }
14:     void FaireMort() const { cout << "Tu dors ?" << endl; }
15:  };
16:
17:  typedef void (Chien::*PFC)()const ;
18:  int main()
19:  {
20:     const int Max = 7;
21:     PFC FonctionsChien[Max] =
22:        {Chien::Crier,
```

```
 23:         Chien::Bouger,
 24:         Chien::Manger,
 25:         Chien::Grogner,
 26:         Chien::Couiner,
 27:         Chien::Jouer,
 28:         Chien::FaireMort };
 29:
 30:     Chien* pChien =0;
 31:     int Methode;
 32:     bool fQuit = false;
 33:
 34:     while (!fQuit)
 35:     {
 36:        cout << "(0)Quitter (1)Crier (2)Bouger ";
36a:       cout << "(3)Manger (4)Grogner ";
 37:        cout << "(5)Couiner (6)Jouer (7)Faire le mort : ";
 38:        std::cin >> Methode;
 39:        if (Methode <= 0 || Methode >= 8)
 40:        {
 41:           fQuit = true;
 42:        }
 43:        else
 44:        {
 45:           pChien = new Chien;
 46:           (pChien->*FonctionsChien[Methode - 1])();
 47:           delete pChien;
 48:        }
 49:     }
 50:     return 0;
 51:  }
```

Ce programme produit le résultat suivant :

```
(0)Quitter (1)Crier (2)Bouger (3)Manger (4)Grogner (5)Couiner
(6)Jouer (7)Faire le mort: 1
Ouah Ouah !
(0)Quitter (1)Crier (2)Bouger (3)Manger (4)Grogner (5)Couiner
(6)Jouer (7)Faire le mort: 4
Grrrrr
(0)Quitter (1)Crier (2)Bouger (3)Manger (4)Grogner (5)Couiner
6)Jouer (7)Faire le mort: 7
Tu dors ?
(0)Quitter (1)Crier (2)Bouger (3)Manger (4)Grogner (5)Couiner
(6)Jouer (7)Faire le mort: 0
```

La classe `Chien` est créée de la ligne 5 à la ligne 15 avec sept fonctions membres renvoyant le même type de résultat et ayant la même signature. `typedef` déclare le type `PFC` à la ligne 17 comme pointeur vers une fonction membre constante de `Chien` sans paramètre et ne renvoyant rien, ce qui correspond aux signatures des sept méthodes de `Chien`.

Le Tableau `FonctionsChien` est déclaré de la ligne 21 à la ligne 28, pour contenir sept pointeurs de fonctions membres analogues, et il est initialisé avec les adresses de ces fonctions.

On demande ensuite à l'utilisateur de choisir une méthode. Sauf s'il choisit Quitter, un objet `Chien` est créé sur le tas, puis appelle la méthode choisie à la ligne 46) Voici une autre ligne que vous pouvez présenter aux programmeurs chevronnés de votre entreprise ; demandez-leur ce qu'elle signifie :

```
(pChien->*FonctionsChien[Methode - 1])();
```

Il s'agit d'un appel à une méthode d'un objet *via* un pointeur de méthode stocké dans un tableau à l'indice `Methode - 1`.

Une fois de plus, cette instruction doit être évitée autant que possible. Si vous l'utilisez, documentez-la minutieusement et essayez d'imaginer une autre manière d'accomplir la tâche.

Faire	*Ne pas faire*
● Appeler des pointeurs sur des fonctions membres sur un objet spécifique de la classe. ● Utiliser `typedef` pour faciliter la relecture des déclarations de pointeurs sur des fonctions membres.	● Utiliser des pointeurs sur des fonctions membres lorsqu'il existe des solutions plus simples. ● Oublier les parenthèses lorsque vous déclarez un pointeur vers une fonction (contrairement à une fonction qui renvoie un pointeur).

Questions-réponses

Q Pourquoi utiliser des données statiques au lieu de données globales ?

R La portée des données statiques est limitée à la classe. Elles ne sont accessibles que *via* un objet de la classe, par un appel utilisant le nom de la classe si elles sont publiques, ou en utilisant une fonction membre statique. La portée des données statiques est celle de leur classe ; grâce aux restrictions d'accès et au typage fort, elles sont donc plus fiables que les données globales.

Q Pourquoi utiliser des fonctions membres statiques au lieu de fonctions globales ?

R Les fonctions membres statiques ont une portée limitée à la classe et ne peuvent être appelées que *via* un objet de la classe ou par une spécification explicite complète (comme `NomClasse::NomFonction()`).

Q Est-il courant d'avoir recours aux pointeurs de fonctions et de fonctions membres ?

R Non. Leur utilisation est spécifique à certaines situations. Nombreux sont les programmes, même complexes, qui n'exploitent pas ces fonctionnalités, mais il peut exister des situations où ils constituent la seule solution.

Testez vos connaissances

1. Une variable membre statique peut-elle être privée ?
2. Déclarez une variable membre statique appelée `saStatique`, de type `int`.
3. Déclarez une fonction statique appelée `Fonction` qui renvoie un entier et ne prend pas de paramètre.
4. Déclarez un pointeur sur une fonction qui renvoie un entier `long` et prend un entier en paramètre.
5. Modifiez le pointeur de la question précédente, de sorte qu'il pointe sur une fonction membre de la classe `Auto`.
6. Déclarez un tableau `Tableau` contenant dix pointeurs du type défini à la Question 5.

Exercices

1. Écrivez un programme qui déclare une classe contenant une variable membre et une variable membre statique. Le constructeur initialisera la variable membre et incrémentera la variable statique. Utilisez le destructeur pour décrémenter la variable membre.
2. En reprenant le programme de l'Exercice 1, écrivez un petit programme de test qui crée trois objets puis affiche leurs variables membres et la variable membre statique. Détruisez ensuite chaque objet et montrez l'effet obtenu sur la variable membre statique.
3. Modifiez le programme de l'exercice précédent pour utiliser une fonction membre statique pour accéder à la variable membre statique qui doit maintenant être privée.
4. Ajoutez un pointeur de fonction membre pour accéder à la donnée membre non statique du programme de l'Exercice 3. Utilisez ce pointeur pour afficher la valeur de ce membre.

5. Dans le programme précédent, ajoutez deux variables membres à la classe. Définissez des fonctions d'accès pour lire leurs valeurs en faisant en sorte qu'elles renvoient le même type de résultat et qu'elles aient la même signature. Utilisez le pointeur de fonction membre pour accéder à ces méthodes.



16

Concepts avancés d'héritage

Au sommaire de ce chapitre

- Caractéristiques et modélisation de l'agrégation (relation *a-un*)
- Caractéristiques et modélisation de la délégation
- Implémentation d'une classe en termes d'une autre
- Utilisation de l'héritage privé

Agrégation

Vous avez vu dans les exemples précédents que les données membres d'une classe peuvent inclure des objets d'autres types de classes, une situation souvent appelée agrégation, ou relation *a-un.*

À titre d'exemple, étudiez des classes comme `Nom` et `Adresse` :

```
Class Nom
{
   // Informations de classe pour Nom
};
Class Adresse
{
   // Informations de classe pour Adresse
};
```

Ces deux classes pourraient être incluses dans une classe `Employe` :

```
Class Employe
{
   Nom     NomEmp;
   Adresse AdresseEmp;
   // Autres éléments de la classe Employe...
}
```

Une classe `Employe` pourrait donc contenir des variables membres pour un nom et pour une adresse (`Employe` *a-un* `Nom` et `Employe` *a-une* `Adresse`).

Le Listing 16.1 présente un exemple plus complexe en implémentant une classe `String` incomplète mais néammoins fonctionnelle. Ce programme ne produit pas de résultat, mais son contenu sera repris dans les listings suivants.

Listing 16.1 : Classe *String*

```
 0:  // Listing 16.1 La classe String
 1:
 2:  #include <iostream>
 3:  #include <String.h>
 4:  using namespace std;
 5:
 6:  class String
 7:  {
 8:  public:
 9:     // constructeurs
10:     String();
11:     String(const char *const);
12:     String(const String &);
13:     ~String();
14:
15:     // opérateurs surchargés
16:     char & operator[](int indice);
17:     char operator[](int indice) const;
```

```
18:     String operator+(const String&);
19:     void operator+=(const String&);
20:     String & operator=(const String &);
21:
22:     // méthodes générales d'accès
23:     int GetLongueur()const { return saLongueur; }
24:     const char * GetString() const { return saChaine; }
25:     // static int CptConstructeur;
26:
27:  private:
28:     String (int);         // constructeur privé
29:     char * saChaine;
30:     unsigned short saLongueur;
31:
32:  };
33:
34:  // le constructeur par défaut crée des chaînes de 0 octet
35:  String::String()
36:  {
37:     saChaine = new char[1];
38:     saChaine[0] = '\0';
39:     saLongueur=0;
40:     // cout << "\tConstructeur par défaut de String \n";
41:     // CptConstructeur++;
42:  }
43:
44:  // constructeur (utilitaire) prive, utilisé seulement par
45:  // les méthodes de la classe pour créer une nouvelle chaîne
46:  // de la taille requise, contenant des caractères nuls.
47:  String::String(int longueur)
48:  {
49:     saChaine = new char[longueur+1];
50:     for (int i = 0; i <= longueur; i++)
51:        saChaine[i] = '\0';
52:     saLongueur = longueur;
53:     // cout << "\tConstructeur de String(int) \n";
54:     // CptConstructeur++;
55:  }
56:
57:  // Convertit un tableau de caractères en chaîne
58:  String::String(const char * const uneChaine)
59:  {
60:     saLongueur = strlen(uneChaine);
61:     saChaine = new char[saLongueur+1];
62:     for (int i = 0; i < saLongueur; i++)
63:        saChaine[i] = uneChaine[i];
64:     saChaine[saLongueur]='\0';
```

```
65:     // cout << "\tConstructeur de String(char*) \n";
66:     // CptConstructeur++;
67:  }
68:
69:  // constructeur de copie
70:  String::String (const String & rhs)
71:  {
72:     saLongueur = rhs.GetLongueur();
73:     saChaine = new char[saLongueur+1];
74:     for (int i = 0; i < saLongueur; i++)
75:        saChaine[i] = rhs[i];
76:     saChaine[saLongueur] = '\0';
77:     // cout << "\tConstructeur de String(String&) \n";
78:     // CptConstructeur++;
79:  }
80:
81:  // destructeur, libère la mémoire
82:  String::~String ()
83:  {
84:     delete [] saChaine;
85:     saLongueur = 0;
86:     // cout << "\tDestructeur de String \n";
87:  }
88:
89:  // opérateur égal, libère la mémoire existante
90:  // puis copie la chaîne et la taille
91:  String& String::operator=(const String & rhs)
92:  {
93:     if (this == &rhs)
94:        return *this;
95:     delete [] saChaine;
96:     saLongueur = rhs.GetLongueur();
97:     saChaine = new char[saLongueur + 1];
98:     for (int i = 0; i < saLongueur; i++)
99:        saChaine[i] = rhs[i];
100:    saChaine[saLongueur] = '\0';
101:    return *this;
102:    // cout << "\tOperator= de String\n";
103: }
104:
105: // opérateur d'indexation non constant, renvoie
106: // une référence sur un caractère pour qu'il puisse
107: // être modifié !
108: char & String::operator[](int indice)
109: {
110:    if (indice > saLongueur)
111:       return saChaine[saLongueur - 1];
```

```
112:     else
113:        return saChaine[indice];
114:  }
115:
116:  // opérateur d'indexation constant pour utilisation
117:  // sur les objets const (voir constructeur de copie !)
118:  char String::operator[](int indice) const
119:  {
120:     if (indice > saLongueur)
121:        return saChaine[saLongueur - 1];
122:     else
123:        return saChaine[indice];
124:  }
125:
126:  // crée une nouvelle chaîne en ajoutant
127:  // la chaîne actuelle à rhs
128:  String String::operator+(const String& rhs)
129:  {
130:     int  totalLongueur = saLongueur + rhs.GetLongueur();
131:     String temp(totalLongueur);
132:     int i, j;
133:     for (i = 0; i < saLongueur; i++)
134:        temp[i] = saChaine[i];
135:     for (j = 0; j < rhs.GetLongueur(); j++, i++)
136:        temp[i] = rhs[j];
137:     temp[totalLongueur]='\0';
138:     return temp;
139:  }
140:
141:  // modifie la chaîne courante, ne renvoie rien
142:  void String::operator+=(const String& rhs)
143:  {
144:     unsigned short rhsLongueur = rhs.GetLongueur();
145:     unsigned short totalLongueur = saLongueur + rhsLongueur;
146:     String  temp(totalLongueur);
147:     int i, j;
148:     for (i = 0; i < saLongueur; i++)
149:        temp[i] = saChaine[i];
150:     for (j = 0; j < rhs.GetLongueur(); j++, i++)
151:        temp[i] = rhs[i - saLongueur];
152:     temp[totalLongueur]='\0';
153:     *this = temp;
154:  }
155:
156:   // int String::CptConstructeur = 0;
```



Placez le code du Listing 16.1 dans un fichier appelé `String.hpp`. *À chaque fois que vous aurez besoin de la classe* `String`, *vous pourrez inclure ce code à l'aide de la directive* `#include "String.hpp"`, *comme dans l'exemple suivant. Les lignes du listing qui ont été commentées seront expliquées au cours de ce chapitre.*

Le Listing 16.1 définit une classe `String` qui ressemble beaucoup à celle du Listing 13.12 du Chapitre 13. La différence importante entre ces deux classes est, qu'ici, les instructions d'affichage des constructeurs et quelques autres fonctions du Listing 13.12 ont été mises en commentaires afin de les désactiver. Ces fonctions seront utilisées dans les exemples suivants.

La ligne 25 déclare la variable membre statique `CptConstructeur`, qui est initialisée à la ligne 156. Cette variable est incrémentée dans chaque constructeur. Tout ceci est, pour le moment, mis en commentaire et sera activé ultérieurement.

Pour des raisons pratiques, l'implémentation de la classe est placée dans le même fichier que sa déclaration. Dans une application réelle, la déclaration se trouverait dans `String.hpp` et son implémentation dans `String.cpp`. Vous ajouteriez alors `String.cpp` à votre programme (à l'aide de `ajouter fichiers` ou d'un *makefile*) et incluriez `String.hpp` dans `String.cpp` à l'aide d'une directive `#include`.

D'ailleurs, dans un vrai programme, vous utiliseriez la classe `String` de la bibliothèque Standard de C++ à la place de cette classe `String` personnalisée.

Le Listing 16.2 décrit une classe `Employe` contenant trois objets `String`. Ceux-ci correspondent au prénom, au nom et à l'adresse d'un employé.

Listing 16.2 : Classe *Employe* et programme principal

```
 0:  // Listing 16.2 Classe Employe et programme principal
 1:  #include "String.hpp"
 2:
 3:  class Employe
 4:  {
 5:  public:
 6:      Employe();
 7:      Employe(char *, char *, char *, long);
 8:      ~Employe();
 9:      Employe(const Employe&);
10:     Employe & operator= (const Employe &);
11:
12:     const String & GetPrenom() const
13:        { return sonPrenom; }
14:     const String & GetNom() const { return sonNom; }
```

```
15:     const String & GetAdresse() const { return sonAdresse; }
16:     long GetSalaire() const { return sonSalaire; }
17:
18:     void SetPrenom(const String & petitNom)
19:        { sonPrenom = petitNom; }
20:     void SetNom(const String & nomFamille)
21:        { sonNom = nomFamille; }
22:     void SetAdresse(const String & adresse)
23:        { sonAdresse = adresse; }
24:     void SetSalaire(long salaire) { sonSalaire = salaire; }
25:  private:
26:     String    sonPrenom;
27:     String    sonNom;
28:     String    sonAdresse;
29:     long      sonSalaire;
30:  };
31:
32:  Employe::Employe():
33:     sonPrenom(""),
34:     sonNom(""),
35:     sonAdresse(""),
36:     sonSalaire(0)
37:  {}
38:
39:  Employe::Employe(char * Prenom, char * Nom,
40:     char * adresse, long salaire):
41:     sonPrenom(Prenom),
42:     sonNom(Nom),
43:     sonAdresse(adresse),
44:     sonSalaire(salaire)
45:  {}
46:
47:  Employe::Employe(const Employe & rhs):
48:     sonPrenom(rhs.GetPrenom()),
49:     sonNom(rhs.GetNom()),
50:     sonAdresse(rhs.GetAdresse()),
51:     sonSalaire(rhs.GetSalaire())
52:  {}
53:
54:  Employe::~Employe() {}
55:
56:  Employe & Employe::operator= (const Employe & rhs)
57:  {
58:     if (this == &rhs)
59:        return *this;
60:
61:     sonPrenom  = rhs.GetPrenom();
```

```
62:     sonNom     = rhs.GetNom();
63:     sonAdresse = rhs.GetAdresse();
64:     sonSalaire = rhs.GetSalaire();
65:
66:     return *this;
67:  }
68:
69:  int main()
70:  {
71:     Employe JB007("J", "B", "Avenue Ian Fleming", 20000);
72:     JB007.SetSalaire(500000);
73:     String Nom("Bond");
74:     JB007.SetNom(Nom);
75:     JB007.SetPrenom("James");
76:
77:     cout << "Nom : ";
78:     cout << JB007.GetPrenom().GetString();
79:     cout << " " << JB007.GetNom().GetString();
80:     cout << ".\nAdresse : ";
81:     cout << JB007.GetAdresse().GetString();
82:     cout << ".\nSalaire : " ;
83:     cout << JB007.GetSalaire();
84:     return 0;
85:  }
```

Ce programme produit le résultat suivant :

```
Nom : James Bond
Adresse : Avenue Ian Fleming
Salaire : 500000
```

Dans ce programme, la classe `Employe` comprend trois objets `String` (lignes 26 à 28) : `sonPrenom`, `sonNom` et `sonAdresse`.

La ligne 71 crée un objet `Employe` nommé `JB007` en lui passant quatre valeurs. La ligne 72 appelle la fonction d'accès `SetSalaire()` de `Employe` avec la valeur littérale `500000`. Dans un programme professionnel, cette valeur serait soit une valeur dynamique (fixée lors de l'exécution), soit une constante.

La ligne 73 crée et initialise l'objet `Nom` à partir d'une chaîne littérale (de type C). Cet objet `String` est passé à la fonction `SetNom()` à la ligne suivante.

La ligne 75 appelle la fonction membre `SetPrenom()` avec une autre chaîne littérale. Si vous examinez attentivement le code de la classe `Employe`, vous remarquerez qu'elle ne dispose pas d'une méthode `SetPrenom()` prenant une chaine de type C en paramètre : la méthode qui porte ce nom attend une référence constante à un objet `String` (ligne 18).

Le compilateur est capable de s'en sortir car il sait construire un objet `String` à partir d'une chaîne de caractères littérale. Il le sait parce que vous lui avez indiqué comment faire à la ligne 11 du Listing 16.1.

Si vous examinez les lignes 78, 79 et 81, vous constaterez quelque chose d'inhabituel. Vous pourriez en effet vous demander pourquoi GetString() a été placé après les appels aux différentes méthodes de la classe `Employe`, comme ici :

```
78:     cout << JB007.GetPrenom().GetString();
```

La méthode `GetPrenom()` de l'objet `JB007` renvoie un objet `String`. Malheureusement, notre objet `String` du Listing 16.1 ne permet pas encore d'utiliser directement l'instruction cout `<<` : nous devons encore passer par une chaîne de type C, ce qui est justement le rôle de la méthode `GetString()`. Nous proposerons une meilleure solution ultérieurement.

Accès aux membres d'une classe agrégée

Une classe qui agrège d'autres objets n'a pas d'accès spécial aux données et fonctions membres de cet objet. Elle a simplement l'accès indiqué. Les objets `Employe`, par exemple, n'ont pas d'accès spécial aux variables membres de `String`. Si l'objet `Employe` "`JB007`" tente d'accéder à la variable membre privée `saLongueur` de sa propre variable `sonPrenom`, le compilateur produira une erreur. Ce n'est cependant pas un gros souci puisque les méthodes d'accès fournissent une interface pour la classe `String` et la classe `Employe` n'a donc pas besoin de connaître les détails de son implémentation, pas plus que de la façon dont sa variable entière, `sonSalaire`, stocke sa valeur.



Les membres agrégés n'ont pas d'accès spécial aux membres de la classe dans laquelle ils sont agrégés. La seule possibilité qu'ils ont d'accéder à l'instance qui les agrège est de leur passer en paramètre une copie du pointeur `this` au moment de leur création, ou plus tard. Dans ce cas, ils auront le même accès à cet objet qu'à un autre.

Contrôle de l'accès aux membres agrégés

La classe `String` fournit l'opérateur + (`operator+`). Le concepteur de la classe `Employe` a verrouillé l'accès de cet opérateur aux objets `Employe` en déclarant que toutes les méthodes d'accès aux objets `String`, comme `GetPrenom()` renvoient une référence constante. `operator+` n'étant pas (et ne pouvant pas être) une méthode constante (puisqu'elle modifie la valeur de l'objet sur lequel elle est appelé), la ligne suivante produirait donc une erreur de compilation :

```
String tampon = JB007.GetPrenom() + JB007.GetNom();
```

`GetPrenom()` renvoie un objet `String` constant, or vous ne pouvez pas appeler `operator+` sur un objet constant.

Pour résoudre ce problème, vous pouvez surcharger la fonction `GetPrenom()` et fournir une version non constante :

```
const String & GetPrenom() const { return sonPrenom; }
String & GetPrenom() { return sonPrenom; }
```

La valeur renvoyée comme la fonction membre ne sont plus constantes. Il n'aurait pas suffit de modifier simplement la valeur renvoyer puisque le résultat d'une fonction n'est pas suffisant pour surcharger une fonction : vous devez également modifier la "constance" de la fonction.

Coût de l'agrégation

Les objets agrégés peuvent pénaliser les performances. À chaque fois qu'un objet `Employe` est créé ou copié, il faut en effet construire tous ses objets `String` agrégés.

Supprimez les barres obliques de toutes les instructions `cout` du Listing 16.1 afin de vous rendre compte du nombre de fois où sont appelés les constructeurs. Le Listing 16.3 reprend le programme précédent en lui ajoutant des instructions d'affichage indiquant les instants où le programme crée des objets. Supprimez les barres obliques du Listing 16.1, puis compilez le Listing 16.3.



Pour compiler ce listing, décommentez les lignes 40, 53, 65, 77, 86 et 102 du Listing 16.1.

Listing 16.3 : Constructeurs de classe agrégée

```
 0:  //Listing 16.3 Constructeurs de classe agrégée
 1:  #include "String.hpp"
 2:
 3:  class Employe
 4:  {
 5:  public:
 6:     Employe();
 7:     Employe(char *, char *, char *, long);
 8:     ~Employe();
 9:     Employe(const Employe&);
10:     Employe & operator= (const Employe &);
11:
12:     const String & GetPrenom() const
13:        { return sonPrenom; }
```

```
14:     const String & GetNom() const { return sonNom; }
15:     const String & GetAdresse() const { return sonAdresse; }
16:     long GetSalaire() const { return sonSalaire; }
17:
18:     void SetPrenom(const String & petitNom)
19:        { sonPrenom = petitNom; }
20:     void SetNom(const String & nomFamille)
21:        { sonNom = nomFamille; }
22:     void SetAdresse(const String & adresse)
23:        { sonAdresse = adresse; }
24:     void SetSalaire(long salaire) { sonSalaire = salaire; }
25:  private:
26:     String    sonPrenom;
27:     String    sonNom;
28:     String    sonAdresse;
29:     long      sonSalaire;
30:  };
31:
32:  Employe::Employe():
33:     sonPrenom(""),
34:     sonNom(""),
35:     sonAdresse(""),
36:     sonSalaire(0)
37:  {}
38:
39:  Employe::Employe(char * Prenom, char * Nom,
40:     char * adresse, long salaire):
41:     sonPrenom(Prenom),
42:     sonNom(Nom),
43:     sonAdresse(adresse),
44:     sonSalaire(salaire)
45:  {}
46:
47:  Employe::Employe(const Employe & rhs):
48:     sonPrenom(rhs.GetPrenom()),
49:     sonNom(rhs.GetNom()),
50:     sonAdresse(rhs.GetAdresse()),
51:     sonSalaire(rhs.GetSalaire())
52:  {}
53:
54:  Employe::~Employe() {}
55:
56:  Employe & Employe::operator= (const Employe & rhs)
57:  {
58:     if (this == &rhs)
59:        return *this;
60:
61:     sonPrenom = rhs.GetPrenom();
62:     sonNom = rhs.GetNom();
```

```
63:     sonAdresse = rhs.GetAdresse();
64:     sonSalaire = rhs.GetSalaire();
65:
66:     return *this;
67:  }
68:
69:  int main()
70:  {
71:     cout << "Création de JB007...\n";
72:     Employe JB007("J", "B", "Avenue Ian Fleming", 20000);
73:     JB007.SetSalaire(20000);
74:     cout << "Appel de SetPrenom() avec char *...\n";
75:     JB007.SetPrenom("James");
76:     cout << "Création du String temporaire Nom...\n";
77:     String Nom("Bond");
78:     JB007.SetNom(Nom);
79:
80:     cout << "Nom : ";
81:     cout << JB007.GetPrenom().GetString();
82:     cout << " " << JB007.GetNom().GetString();
83:     cout << "\nAdresse : ";
84:     cout << JB007.GetAdresse().GetString();
85:     cout << "\nSalaire : " ;
86:     cout << JB007.GetSalaire();
87:     cout << endl;
88:     return 0;
89:  }
```

Ce programme produit le résultat suivant :

```
 1: Création de JB007...
 2:    Constructeur de String(char*)
 3:    Constructeur de String(char*)
 4:    Constructeur de String(char*)
 5: Appel de SetPrenom() avec char * ...
 6:    Constructeur de String(char*)
 7:    Destructeur de String
 8: Création de la chaîne temporaire Nom
 9:    Constructeur de String(char*)
10: Nom : James Bond
11: Adresse : Avenue Ian Fleming
12: Salaire : 20000
13:   Destructeur de String
14:   Destructeur de String
15:   Destructeur de String
16:   Destructeur de String
```

Les déclarations de classe sont communes aux Listing 16.1 et 16.2. Toutefois, les instructions `cout` ne figurent plus en commentaires. Le résultat a été numéroté pour faciliter son analyse.

Le message `Création de JB007...` est affiché par la ligne 71 du Listing 16.3 (ligne 1 du résultat). L'objet est alors créé avec quatre paramètres, comme l'indique le résultat, les trois premiers étant des `String`. Le constructeur de `String` est appelé trois fois, comme prévu.

Le prénom du salarié est alors initialisé (ligne 75) comme l'indique le résultat. Cette variable temporaire est détruite immédiatement après l'affectation.

La ligne 77 crée un objet `String`. Le programmeur utilise explicitement l'opération faite implicitement par le compilateur à l'instruction précédente. Le résultat indique que le constructeur est appelé, mais pas le destructeur. En effet, l'objet ne sera supprimé que lorsqu'il deviendra hors de portée, à la fin de la fonction.

Les objets `String` de l'objet `Employe` sont supprimées lorsque l'objet lui-même devient hors de portée. La chaîne `Nom` disparaît également (lignes 81 à 87).

Copie par valeur

Vous avez pu constater, dans le Listing 16.3, que la création d'un objet `Employe` entraîne l'appel de cinq constructeurs de `String`. Dans le Listing 16.4, nous allons reprendre la fonction principale de ce programme mais, cette fois-ci, en activant la variable membre statique `CptConstructeur`.

L'examen du Listing 16.1 montre que `CptConstructeur` est incrémentée à chaque fois qu'un constructeur de chaîne est appelé. Dans le Listing 16.4, le programme principal appelle les fonctions d'affichage en leur passant l'objet `Employe`, d'abord par référence puis par valeur. `CptConstructeur` enregistre le nombre d'objets `String` créés lorsque l'objet `Employe` est passé en paramètre.



Pour compiler ce programme, laissez les lignes que vous avez décommentées dans le Listing 16.1, et décommentez les lignes 41, 54, 66, 78 et 156.

Listing 16.4 : Passage de paramètres par valeur

```
0:  // Listing 16.4 Passage par valeur
1:  #include "String.hpp"
2:
3:  class Employe
4:  {
```

```
 5:  public:
 6:     Employe();
 7:     Employe(char *, char *, char *, long);
 8:     ~Employe();
 9:     Employe(const Employe&);
10:     Employe & operator= (const Employe &);
11:
12:     const String & GetPrenom() const
13:        { return sonPrenom; }
14:     const String & GetNom() const { return sonNom; }
15:     const String & GetAdresse() const { return sonAdresse; }
16:     long GetSalaire() const { return sonSalaire; }
17:
18:     void SetPrenom(const String & petitNom)
19:        { sonPrenom = petitNom; }
20:     void SetNom(const String & nomFamille)
21:        { sonNom = nomFamille; }
22:     void SetAdresse(const String & adresse)
23:        { sonAdresse = adresse; }
24:     void SetSalaire(long salaire) { sonSalaire = salaire; }
25:  private:
26:     String    sonPrenom;
27:     String    sonNom;
28:     String    sonAdresse;
29:     long      sonSalaire;
30:  };
31:
32:  Employe::Employe():
33:     sonPrenom(""),
34:     sonNom(""),
35:     sonAdresse(""),
36:     sonSalaire(0)
37:  {}
38:
39:  Employe::Employe(char * Prenom, char * Nom,
40:     char * adresse, long salaire):
41:     sonPrenom(Prenom),
42:     sonNom(Nom),
43:     sonAdresse(adresse),
44:     sonSalaire(salaire)
45:  {}
46:
47:  Employe::Employe(const Employe & rhs):
48:     sonPrenom(rhs.GetPrenom()),
49:     sonNom(rhs.GetNom()),
50:     sonAdresse(rhs.GetAdresse()),
51:     sonSalaire(rhs.GetSalaire())
```

```
52:  {}
53:
54:  Employe::~Employe() {}
55:
56:  Employe & Employe::operator= (const Employe & rhs)
57:  {
58:     if (this == &rhs)
59:        return *this;
60:
61:     sonPrenom = rhs.GetPrenom();
62:     sonNom = rhs.GetNom();
63:     sonAdresse = rhs.GetAdresse();
64:     sonSalaire = rhs.GetSalaire();
65:
66:     return *this;
67:  }
68:
69:  void FctAffichage(Employe);
70:  void rFctAffichage(const Employe&);
71:
72:  int main()
73:  {
74:     Employe JB007("J", "B", "Avenue Ian Fleming", 20000);
75:     JB007.SetSalaire(20000);
76:     JB007.SetPrenom("James");
77:     String Nom("Bond");
78:     JB007.SetNom(Nom);
79:
80:     cout << "Total constructeurs : " ;
81:     cout << String::CptConstructeur << endl;
82:     rFctAffichage (JB007);
83:     cout << "Total constructeurs : ";
84:     cout << String::CptConstructeur << endl;
85:     FctAffichage(JB007);
86:     cout << "Total constructeurs : ";
87:     cout << String::CptConstructeur << endl;
88:     return 0;
89:  }
90:  void FctAffichage(Employe JB007)
91:  {
92:     cout << "Nom : ";
93:     cout << JB007.GetPrenom().GetString();
94:     cout << " " << JB007.GetNom().GetString();
95:     cout << ".\nAdresse : ";
96:     cout << JB007.GetAdresse().GetString();
97:     cout << ".\nSalaire : " ;
98:     cout << JB007.GetSalaire();
99:     cout << endl;
```

```
100:  }
101:
102:  void rFctAffichage(const Employe& JB007)
103:  {
104:     cout << "Nom : ";
105:     cout << JB007.GetPrenom().GetString();
106:     cout << " " << JB007.GetNom().GetString();
107:     cout << "\nAdresse : ";
108:     cout << JB007.GetAdresse().GetString();
109:     cout << "\nSalaire : " ;
110:     cout << JB007.GetSalaire();
111:     cout << endl;
112:  }
```

Ce programme produit le résultat suivant :

```
        Constructeur de String(char*)
        Constructeur de String(char*)
        Constructeur de String(char*)
        Constructeur de String(char*)
        Destructeur de String
        Constructeur de String(char*)
Total constructeurs : 5
Nom : James Bond
Adresse : Avenue Ian Fleming
Salaire : 20000
Total constructeurs : 5
        Constructeur de String(String&)
        Constructeur de String(String&)
        Constructeur de String(String&)
Nom : James Bond
Adresse: Avenue Ian Fleming.
Salaire: 20000
        Destructeur de String
        Destructeur de String
        Destructeur de String
Total constructeurs : 8
        Destructeur de String
        Destructeur de String
        Destructeur de String
        Destructeur de String
```

Le résultat montre qu'il faut cinq objets `String` pour créer un objet `Employe` (ligne 74). Lorsque celui-ci est passé, ligne 82, en référence à `rFctAffichage()`, le programme ne crée pas d'objets `Employe` ni d'objets `String` supplémentaires (ils sont passés également par référence). On le voit au début du résultat, car que le compte reste à 5 et qu'aucun constructeur n'est appelé

À la ligne 85, l'objet `Employe` est passé par valeur à la fonction `FctAffichage()`, ce qui entraîne la copie de cet objet et la création de trois objets `String` supplémentaires (par appel du constructeur de copie).

Héritage et Délégation/Agrégation

Il arrive qu'une classe ait besoin des fonctionnalités d'une autre classe. Supposons, par exemple, que vous souhaitiez créer une classe `Catalogue`. Par définition, un catalogue comprend des références uniques à des objets (des numéros de pièces, par exemple). Il est donc impossible de créer des doublons dans la classe `Catalogue`.

Vous pourriez faire en sorte que `Catalogue` contienne une classe `ListePieces` et qu'elle délègue la gestion de la liste chaînée des pièces à son objet agrégé `ListePieces`.

Il serait également possible faire dériver `Catalogue` de `ListePieces` et donc d'en hériter ses fonctionnalités. Cependant, l'héritage public établit une relation *est-un* (voir Chapitre 11) et vous devez vous demander si un `Catalogue` est vraiment un type de `ListesPieces.`

Une façon de répondre à cette question consiste à admettre que `ListePieces` est la classe de base de `Catalogue` et à se poser les questions suivantes :

1. La classe de base contient-elle un ou plusieurs objets qui ne doivent pas figurer dans la classe dérivée ? `ListePieces` contient-elle, par exemple, des méthodes qui n'ont rien à faire dans la classe `Catalogue` ? Si la réponse est oui, renoncez à une relation d'héritage public.
2. La classe dérivée peut-elle être associée à plusieurs objets de la classe de base ? Un objet `Catalogue`, par exemple, peut-il être avoir besoin de plusieurs objets `ListePieces` pour accomplir sa tâche ? Si la réponse est oui, préférez l'agrégation.
3. Devez-vous hériter de la classe de base pour bénéficier des fonctions virtuelles et des membres à accès protégé ? Si la réponse est oui, créez une relation d'héritage public ou privé.

En fonction de vos réponses, vous pouvez choisir entre l'héritage public (relation *est-un*), l'héritage privé (qui sera expliqué plus loin) ou l'agrégation (relation *a-un*).

Quelques termes

Nous utilisons ici plusieurs termes nouveaux :

- **Agrégation.** Un objet déclaré membre d'une autre classe est contenu dans cette classe. Aussi appelé relation *a-un*.

- **Délégation.** Utilisation des membres d'une classe agrégée pour exécuter des méthodes pour le compte de la classe contenante.
- **Implémentation en termes de.** Créer une classe à partir des fonctionnalités d'une autre, sans avoir recours à l'héritage public (en utilisant, par exemple, un héritage protégé ou privé).

Délégation

Pourquoi ne pas dériver `Catalogue` de `ListePieces` ? `Catalogue` n'est pas une `ListePieces` puisque cette dernière représente une collection triée dont les éléments peuvent se répéter. Un `Catalogue`, par contre, contient des entrées uniques stockées pêle-mêle. Son cinquième élément ne porte pas le numéro 5, mais n'importe quel numéro de pièce.

Il aurait été certainement possible d'hériter publiquement de `ListePieces`, puis de redéfinir la fonction `Inserer()` et les opérateurs d'indexation pour qu'ils se comportent de façon appropriée, mais on modifierait alors l'essence même de la classe `ListePieces`. Il est préférable de construire un `Catalogue` sans opérateur d'indexation et qui n'autorise pas les doublons, et définir l'opérateur + afin qu'il concatène deux ensembles.

La première façon de réaliser tout cela consiste à utiliser l'agrégation : la classe `Catalogue` déléguera la gestion de la liste des pièces à un objet `ListePieces` agrégé. C'est ce qui est présenté dans le Listing 16.5.

Listing 16.5 : Délégation à une *ListePieces* agrégée

```
 0:  // Listing 16.5 Délégation à une liste agrégée
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  // *************** Piece ************
 6:
 7:  // Classe de base abstraite de pièces
 8:  class Piece
 9:  {
10:  public:
11:     Piece(): saNumPiece(1) {}
12:     Piece(int NumPiece):
13:       saNumPiece(NumPiece) {}
14:     virtual ~Piece(){}
15:     int GetNumPiece() const
16:        { return saNumPiece; }
17:     virtual void Afficher() const = 0;
18:  private:
```

```
19:     int saNumPiece;
20:  };
21:
22:  // implémentation d'une fonction virtuelle pure
23:  // pour chaîner les classes dérivées
24:  void Piece::Afficher() const
25:  {
26:     cout << "\nNuméro de pièce : " << saNumPiece << endl;
27:  }
28:
29:  // *************** Piece Auto ************
30:
31:  class PieceAuto : public Piece
32:  {
33:  public:
34:     PieceAuto(): sonAnneeAuto(97) {}
35:     PieceAuto(int annee, int NumPiece);
36:     virtual void Afficher() const
37:     {
38:     Piece::Afficher();
39:     cout << "Millésime : ";
40:     cout << sonAnneeAuto << endl;
41:     }
42:  private:
43:     int sonAnneeAuto;
44:  };
45:
46:  PieceAuto::PieceAuto(int annee, int NumPiece):
47:     sonAnneeAuto(annee),
48:     Piece(NumPiece)
49:  {}
50:
51:
52:  // *************** Piece Avion ************
53:
54:  class PieceAvion : public Piece
55:  {
56:  public:
57:     PieceAvion(): saRefMoteur(1) {};
58:     PieceAvion
59:      (int RefMoteur, int NumPiece);
60:     virtual void Afficher() const
61:     {
62:     Piece::Afficher();
63:     cout << "Référence moteur : ";
64:     cout << saRefMoteur << endl;
65:     }
```

```
 66:  private:
 67:     int saRefMoteur;
 68:  };
 69:
 70:  PieceAvion::PieceAvion
 71:  (int RefMoteur, int NumPiece):
 72:     saRefMoteur(RefMoteur),
 73:     Piece(NumPiece)
 74:  {}
 75:
 76:  // *************** Noeud Piece ************
 77:  class NoeudPiece
 78:  {
 79:  public:
 80:     NoeudPiece(Piece*);
 81:     ~NoeudPiece();
 82:     void SetSuivant(NoeudPiece * noeud)
 83:        { sonSuivant = noeud; }
 84:     NoeudPiece * GetSuivant () const;
 85:     Piece * GetPiece () const;
 86:  private:
 87:     Piece *saPiece ;
 88:     NoeudPiece * sonSuivant;
 89:  };
 90:  // NoeudPiece : implementations...
 91:
 92:  NoeudPiece::NoeudPiece(Piece* pPiece):
 93:     saPiece (pPiece),
 94:     sonSuivant(0)
 95:  {}
 96:
 97:  NoeudPiece::~NoeudPiece()
 98:  {
 99:     delete saPiece ;
100:     saPiece  = 0;
101:     delete sonSuivant;
102:     sonSuivant = 0;
103:  }
104:
105:  // Renvoie NULL si pas de NoeudPiece suivant
106:  NoeudPiece * NoeudPiece::GetSuivant () const
107:  {
108:     return sonSuivant;
109:  }
110:
111:  Piece * NoeudPiece::GetPiece () const
112:  {
```

```
113:     if (saPiece )
114:        return saPiece ;
115:     else
116:        return NULL; // erreur
117:  }
118:
119:
120:
121:  // ***************** Liste Pieces ************
122:  class ListePieces
123:  {
124:  public:
125:     ListePieces();
126:     ~ListePieces();
127:     // nécessite constructeur de copie et opérateur Egal !
128:     void    Traiter(void (Piece::*f)()const) const;
129:     Piece*  Rechercher(int & position, int NumPiece)  const;
130:     Piece*  GetPremier() const;
131:     void    Inserer(Piece *);
132:     Piece*  operator[](int) const;
133:     int     GetTotal() const { return sonTotal; }
134:     static   ListePieces& GetListePiecesGlobale()
135:     {
136:     return  ListePiecesGlobale;
137:     }
138:  private:
139:     NoeudPiece * pTete;
140:     int sonTotal;
141:     static ListePieces ListePiecesGlobale;
142:  };
143:
144:  ListePieces ListePieces::ListePiecesGlobale;
145:
146:
147:  ListePieces::ListePieces():
148:     pTete(0),
149:     sonTotal(0)
150:  {}
151:
152:  ListePieces::~ListePieces()
153:  {
154:     delete pTete;
155:  }
156:
157:  Piece*   ListePieces::GetPremier() const
158:  {
159:     if (pTete)
```

```
160:        return pTete->GetPiece ();
161:     else
162:        return NULL;  // interception d'erreur
163:  }
164:
165:  Piece *  ListePieces::operator[](int indice) const
166:  {
167:     NoeudPiece* pNoeud = pTete;
168:
169:     if (!pTete)
170:        return NULL; // interception d'erreur
171:
172:     if (indice > sonTotal)
173:        return NULL; // erreur
174:
175:     for (int i=0; i < indice; i++)
176:        pNoeud = pNoeud->GetSuivant ();
177:
178:     return pNoeud->GetPiece ();
179:  }
180:
181:  Piece* ListePieces::Rechercher(
182:     int & position,
183:     int NumPiece) const
184:  {
185:     NoeudPiece * pNoeud = 0;
186:     for (pNoeud = pTete, position = 0;
187:           pNoeud!=NULL;
188:           pNoeud = pNoeud->GetSuivant (), position++)
189:     {
190:        if (pNoeud->GetPiece ()->GetNumPiece() == NumPiece)
191:           break;
192:     }
193:     if (pNoeud == NULL)
194:        return NULL;
195:     else
196:        return pNoeud->GetPiece ();
197:  }
198:
199:  void ListePieces::Traiter(void (Piece::*fct)()const) const
200:  {
201:     if (!pTete)
202:        return;
203:     NoeudPiece* pNoeud = pTete;
204:     do
205:        (pNoeud->GetPiece ()->*fct)();
206:     while ((pNoeud = pNoeud->GetSuivant ()) != 0);
```

```
207:  }
208:
209:  void ListePieces::Inserer(Piece* pPiece)
210:  {
211:     NoeudPiece * pNoeud = new NoeudPiece(pPiece);
212:     NoeudPiece * pCourant = pTete;
213:     NoeudPiece * pSuivant = 0;
214:
215:     int Nouveau =  pPiece->GetNumPiece();
216:     int Suivant = 0;
217:     sonTotal++;
218:
219:     if (!pTete)
220:     {
221:        pTete = pNoeud;
222:        return;
223:     }
224:
225:     // si celui-ci est plus petit que Tete
226:     // il devient le nouveau noeud de Tete
227:     if (pTete->GetPiece ()->GetNumPiece() > Nouveau)
228:     {
229:        pNoeud->SetSuivant(pTete);
230:        pTete = pNoeud;
231:        return;
232:     }
233:
234:     for (;;)
235:     {
236:        // si pas de suivant, ajouter celui-ci
237:        if (!pCourant->GetSuivant ())
238:        {
239:           pCourant->SetSuivant(pNoeud);
240:           return;
241:        }
242:
243:        // s'il vient après celui-ci et avant le suivant
244:        // l'insérer ici, sinon lire le suivant
245:        pSuivant = pCourant->GetSuivant ();
246:        Suivant = pSuivant->GetPiece ()->GetNumPiece();
247:        if (Suivant > Nouveau)
248:        {
249:           pCourant->SetSuivant(pNoeud);
250:           pNoeud->SetSuivant(pSuivant);
251:           return;
252:        }
253:        pCourant = pSuivant;
```

```
254:     }
255:  }
256:
257:  class Catalogue
258:  {
259:  public:
260:     void Inserer(Piece *);
261:     int Existe(int NumPiece);
262:     Piece * Lire(int NumPiece);
263:   //  operator+(const Catalogue &);
264:     void AfficherTout()
264a:      { saListePieces.Traiter(Piece::Afficher); }
265:  private:
266:     ListePieces saListePieces;
267:  };
268:
269:  void Catalogue::Inserer(Piece * nouvPiece )
270:  {
271:     int NumPiece =  nouvPiece ->GetNumPiece();
272:     int indice;
273:
274:     if (!saListePieces.Rechercher(indice, NumPiece))
275:     {
276:        saListePieces.Inserer(nouvPiece );
277:     }
278:     else
279:     {
280:        cout << NumPiece << " était la ";
281:        switch (indice)
282:        {
283:        case 0:  cout << "première "; break;
284:        case 1:  cout << "deuxième "; break;
285:        case 2:  cout << "troisième "; break;
286:        default: cout << indice + 1 << "e";
287:        }
288:        cout << "entrée. Refusée !" << endl;
289:     }
290:  }
291:
292:  int Catalogue::Existe(int NumPiece)
293:  {
294:     int indice;
295:     saListePieces.Rechercher(indice, NumPiece);
296:     return indice;
297:  }
298:
299:  Piece * Catalogue::Lire(int NumPiece)
```

```
300:  {
301:     int indice;
302:    Piece * laPiece = saListePieces.Rechercher(indice,
302a:                                                NumPiece);
303:     return laPiece;
304:  }
305:
306:
307:  int main()
308:  {
309:     Catalogue cat;
310:     Piece * pPiece = 0;
311:     int NumPiece;
312:     int valeur;
313:     int choix = 99;
314:
315:     while (choix!= 0)
316:     {
317:        cout << "(0)Quitter (1)Auto (2)Avion - Votre choix : ";
318:        cin >> choix;
319:
320:        if (choix!= 0)
321:        {
322:           cout << "Entrez un numéro de pièce : ";
323:           cin >>  NumPiece;
324:
325:           if (choix == 1)
326:           {
327:              cout << "Millésime ? ";
328:              cin >> valeur;
329:              pPiece = new PieceAuto(valeur, NumPiece);
330:           }
331:           else
332:           {
333:              cout << "Référence moteur ? ";
334:              cin >> valeur;
335:              pPiece = new PieceAvion(valeur, NumPiece);
336:           }
337:           cat.Inserer(pPiece);
338:        }
339:     }
340:     cat.AfficherTout();
341:     return 0;
342:  }
```

Ce programme produit le résultat suivant :

```
(0)Quitter (1)Auto (2)Avion - Votre choix :  1
Entrez un numéro de pièce : 1234
Millésime ? 94
(0)Quitter (1)Auto (2)Avion - Votre choix :  1
Entrez un numéro de pièce : 4434
Millésime ? 93
(0)Quitter (1)Auto (2)Avion - Votre choix :  1
Entrez un numéro de pièce : 1234
Millésime ? 94
1234 était la première entrée. Refusée !
(0)Quitter (1)Auto (2)Avion - Votre choix :  1
Entrez un numéro de pièce : 2345
Millésime ? 93
(0)Quitter (1)Auto (2)Avion - Votre choix :  0

Numéro de pièce : 1234
Millésime : 94

Numéro de pièce : 2345
Millésime : 93

Numéro de pièce : 4434
Millésime : 93
```



Certains compilateurs ne peuvent traiter la ligne 264, bien qu'elle soit légale en C++. Si votre compilateur signale une erreur, remplacez la ligne par :

```
264: void AfficherTout() { laListePieces.Traiter(&Piece::Afficher); }
```

(Remarquez l'ajout de l'esperluette devant `Piece::Afficher`.) Si le problème est résolu de cette façon, appelez immédiatement le fournisseur de votre compilateur pour protester.

Une nouvelle classe `Catalogue` est déclarée aux lignes 257 à 267. Elle délègue la gestion de sa liste à la liste `ListePieces` (ligne 265). En d'autres termes, `Catalogue` est implémentée en termes de `ListePieces`.

Les clients de `Catalogue` n'ont pas directement accès à `ListePieces` car elle est déclarée comme membre privé. L'accès à cette classe passe par l'interface de `Catalogue`, ce qui modifie considérablement son comportement. La méthode `Catalogue::Inserer()`, par exemple, ne permet pas de dupliquer les entrées dans `ListePieces`.

L'implémentation de `Catalogue::Inserer()` est réalisée à partir de la ligne 269. La valeur `saNumPiece` de la pièce passée en paramètre est transmise à la méthode

`Rechercher()` de `ListePieces` à la ligne 274. Si elle n'a pas été trouvée, la nouvelle pièce est insérée dans la liste (ligne 276) ; sinon, un message d'erreur apparaît (à partir de la ligne 280).

Vous remarquerez que `Catalogue` réalise l'insertion en appelant la méthode `Inserer()` sur sa variable membre `saListePieces`, qui est une `ListePieces`. En fait, les mécanismes d'insertion et de gestion de la liste chaînée, ainsi que les opérations de recherche et d'extraction de la liste chaînée sont encapsulés dans le membre `ListePieces` agrégé à `Catalogue`. Il n'y a aucune raison que `Catalogue` reproduise ce code ; elle peut tirer parti d'une interface bien conçue.

C'est l'essence de la réutilisabilité du code en C++ : `Catalogue` peut réutiliser le code de `ListePieces`, ce qui évite au concepteur de `Catalogue` de devoir connaître les détails d'implémentation de `ListePieces`. L'interface de cette dernière (c'est-à-dire sa déclaration) fournit toutes les informations nécessaires au concepteur de `Catalogue`.

Héritage privé

Pour accéder aux membres protégés de `ListePieces` (qui n'en possède pas ici) ou pour redéfinir des méthodes de `ListePieces`, `Catalogue` doit hériter publiquement de `ListePieces`.

`Catalogue` n'étant pas un objet `ListePieces` et comme vous ne souhaitez pas exposer l'ensemble des fonctionnalités de `ListePieces` aux clients de `Catalogue`, vous devez utiliser un héritage privé. L'héritage privé permet en effet d'hériter d'une autre classe tout en maintenant privée la partie interne de cette classe pour la classe dérivée.

La première chose à savoir sur l'héritage privé est que toutes les variables et les fonctions membres de la classe de base sont traitées comme si elles avaient été déclarées privées, quel que soit leur déclaration d'accès dans la classe de base. Par conséquent, chaque fonction héritée de `ListePieces` est inaccessible à toute fonction qui n'est pas membre de `Catalogue`. Ce point est essentiel : l'héritage privé n'implique pas l'héritage de l'interface, seulement de l'implémentation.

La classe `ListePieces` est invisible pour les clients de `Catalogue`. La totalité de son interface leur est inaccessible et ils ne peuvent appeler aucune de ses méthodes. Ils peuvent appeler les méthodes de `Catalogue` qui, elles, ont accès à tous les objets de `ListePieces` car `Catalogue` dérive de `ListePieces`. Le point important, ici, est que `Catalogue` *n'est pas* une `ListePieces` comme cela aurait été le cas avec un héritage public. `Catalogue` est implémentée en termes d'une `ListePieces`, comme cela aurait été le cas avec une agrégation. L'héritage privé est simplement une question de confort.

La Listing 16.6 montre comment utiliser l'héritage privé en récrivant la classe `Catalogue` pour qu'elle hérite en privé de `ListePieces`.

Listing 16.6 : Héritage privé

```
 0:  //Listing 16.6 démonstration de l'héritage privé
 1:  #include <iostream>
 2:  using namespace std;
 3:
 4:  // *************** Piece ************
 5:
 6:  // Classe de base abstraite de pièces
 7:  class Piece
 8:  {
 9:  public:
10:     Piece():saNumPiece(1) {}
11:     Piece(int NumPiece):
12:       saNumPiece(NumPiece){}
13:     virtual ~Piece(){}
14:     int GetNumPiece() const
15:        { return saNumPiece; }
16:     virtual void Afficher() const = 0;
17:  private:
18:     int saNumPiece;
19:  };
20:
21:  // implémentation d'une fonction virtuelle pure
22:  // pour chaîner les classes dérivées
23:  void Piece::Afficher() const
24:  {
25:     cout << "\nNuméro de pièce : " << saNumPiece << endl;
26:  }
27:
28:  // *************** Pièce Auto ************
29:
30:  class PieceAuto : public Piece
31:  {
32:  public:
33:     PieceAuto():sonAnneeAuto(97){}
34:     PieceAuto(int annee, int NumPiece);
35:     virtual void Afficher() const
36:     {
37:        Piece::Afficher();
38:        cout << "Millésime : ";
39:        cout << sonAnneeAuto << endl;
40:     }
41:  private:
```

```
42:     int sonAnneeAuto;
43:  };
44:
45:  PieceAuto::PieceAuto(int annee, int NumPiece):
46:     sonAnneeAuto(annee),
47:     Piece(NumPiece)
48:  {}
49:
50:
51:  // ***************** Pièce Avion************
52:
53:  class PieceAvion : public Piece
54:  {
55:  public:
56:     PieceAvion():saRefMoteur(1){};
57:     PieceAvion(int RefMoteur, int NumPiece);
58:     virtual void Afficher() const
59:     {
60:        Piece::Afficher();
61:        cout << "Référence moteur : ";
62:        cout << saRefMoteur << endl;
63:     }
64:  private:
65:     int saRefMoteur;
66:  };
67:
68:  PieceAvion::PieceAvion
69:  (int RefMoteur, int NumPiece):
70:     saRefMoteur(RefMoteur),
71:     Piece(NumPiece)
72:  {}
73:
74:  // ***************** Noeud Pièce ************
75:  class NoeudPiece
76:  {
77:  public:
78:     NoeudPiece (Piece*);
79:     ~NoeudPiece();
80:     void SetSuivant(NoeudPiece * noeud)
81:        { sonSuivant = noeud; }
82:     NoeudPiece * GetSuivant () const;
83:     Piece * GetPiece () const;
84:  private:
85:     Piece *saPiece ;
86:     NoeudPiece * sonSuivant;
87:  };
88:  // NoeudPiece : implémentations...
```

```
 89:
 90:  NoeudPiece::NoeudPiece(Piece* pPiece):
 91:     saPiece (pPiece),
 92:     sonSuivant(0)
 93:  {}
 94:
 95:  NoeudPiece::~NoeudPiece()
 96:  {
 97:     delete saPiece ;
 98:     saPiece  = 0;
 99:     delete sonSuivant;
100:    sonSuivant = 0;
101:  }
102:
103:  // Renvoie NULL si pas de NoeudPiece suivant
104:  NoeudPiece * NoeudPiece::GetSuivant () const
105:  {
106:     return sonSuivant;
107:  }
108:
109:  Piece * NoeudPiece::GetPiece () const
110:  {
111:     if (saPiece )
112:        return saPiece ;
113:     else
114:        return NULL; // erreur
115:  }
116:
117:
118:
119:  // ***************** Liste Pièces ************
120:  class ListePieces
121:  {
122:  public:
123:     ListePieces();
124:     ~ListePieces();
125:     // nécessite constructeur de copie et opérateur égal!
126:     void     Traiter(void (Piece::*f)()const) const;
127:     Piece*   Rechercher(int & position, int NumPiece)  const;
128:     Piece*   GetPremier() const;
129:     void     Inserer(Piece *);
130:     Piece*   operator[](int) const;
131:     int      GetTotal() const { return sonTotal; }
132:     static   ListePieces& GetListePiecesGlobale()
133:     {
134:        return  ListePiecesGlobale;
135:     }
```

```
136:  private:
137:     NoeudPiece * pTete;
138:     int sonTotal;
139:     static ListePieces ListePiecesGlobale;
140:  };
141:
142:  ListePieces ListePieces::ListePiecesGlobale;
143:
144:
145:  ListePieces::ListePieces():
146:     pTete(0),
147:     sonTotal(0)
148:  {}
149:
150:  ListePieces::~ListePieces()
151:  {
152:     delete pTete;
153:  }
154:
155:  Piece*   ListePieces::GetPremier() const
156:  {
157:     if (pTete)
158:        return pTete->GetPiece ();
159:     else
160:        return NULL;  // interception d'erreur
161:  }
162:
163:  Piece *  ListePieces::operator[](int indice) const
164:  {
165:     NoeudPiece* pNoeud = pTete;
166:
167:     if (!pTete)
168:        return NULL; // interception d'erreur
169:
170:     if (indice > sonTotal)
171:        return NULL; //  erreur
172:
173:     for (int i=0; i < indice; i++)
174:        pNoeud = pNoeud->GetSuivant ();
175:
176:     return   pNoeud->GetPiece ();
177:  }
178:
179:  Piece* ListePieces::Rechercher(int & position,
179a:                                 int NumPiece)  const
180:  {
181:     NoeudPiece * pNoeud = 0;
```

```
182:     for (pNoeud = pTete, position = 0;
183:        pNoeud != NULL;
184:     pNoeud = pNoeud->GetSuivant (), position++)
185:     {
186:        if (pNoeud->GetPiece ()->GetNumPiece() == NumPiece)
187:        break;
188:     }
189:     if (pNoeud == NULL)
190:        return NULL;
191:     else
192:        return pNoeud->GetPiece();
193:  }
194:
195:  void ListePieces::Trainer(void (Piece::*fct)()const) const
196:  {
197:     if (!pTete)
198:        return;
199:     NoeudPiece* pNoeud = pTete;
200:     do
201:        (pNoeud->GetPiece()->*fct)();
202:     while ((pNoeud = pNoeud->GetSuivant ()) != 0);
203:  }
204:
205:  void ListePieces::Inserer(Piece* pPiece)
206:  {
207:     NoeudPiece * pNoeud = new NoeudPiece(pPiece);
208:     NoeudPiece * pCourant = pTete;
209:     NoeudPiece * pSuivant = 0;
210:
211:     int Nouveau =  pPiece->GetNumPiece();
212:     int Suivant = 0;
213:     sonTotal++;
214:
215:     if (!pTete)
216:     {
217:        pTete = pNoeud;
218:        return;
219:     }
220:
221:     // si celui-ci est plus petit que Tete
222:     // il devient le nouveau noeud de Tete
223:     if (pTete->GetPiece ()->GetNumPiece() > Nouveau)
224:     {
225:        pNoeud->SetSuivant(pTete);
226:        pTete = pNoeud;
227:        return;
228:     }
```

```
229:
230:     for (;;)
231:     {
232:        // si pas de suivant, ajouter celui-ci
233:        if (!pCourant->GetSuivant ())
234:        {
235:           pCourant->SetSuivant(pNoeud);
236:           return;
237:        }
238:
239:        // s'il vient après celui-ci et avant le suivant
240:        // l'insérer ici, sinon lire le suivant
241:        pSuivant = pCourant->GetSuivant();
242:        Suivant = pSuivant->GetPiece()->GetNumPiece();
243:        if (Suivant > Nouveau)
244:        {
245:           pCourant->SetSuivant(pNoeud);
246:           pNoeud->SetSuivant(pSuivant);
247:           return;
248:        }
249:        pCourant = pSuivant;
250:     }
251:  }
252:
253:  class Catalogue : private ListePieces
254:  {
255:  public:
256:     void Inserer(Piece *);
257:     int Existe(int NumPiece);
258:     Piece * Lire(int NumPiece);
259:     operator+(const Catalogue &);
260:     void AfficherTout() { Traiter(Piece::Afficher); }
261:  private:
262:  };
263:
264:  void Catalogue::Inserer(Piece * nouvPiece )
265:  {
266:     int NumPiece = nouvPiece ->GetNumPiece();
267:     int indice;
268:
269:     if (!Rechercher(indice, NumPiece))
270:     {
271:        ListePieces::Inserer(nouvPiece);
272:     }
273:     else
274:     {
275:        cout << NumPiece << " était la ";
```

```
276:        switch (indice)
277:        {
278:        case 0:  cout << "première "; break;
279:        case 1:  cout << "deuxième"; break;
280:        case 2:  cout << "troisième"; break;
281:        default: cout << indice + 1 << "e";
282:        }
283:        cout << "entrée. Refusée !" << endl;
284:     }
285:  }
286:
287:  int Catalogue::Existe(int NumPiece)
288:  {
289:     int indice;
290:     Rechercher(indice, NumPiece);
291:     return indice;
292:  }
293:
294:  Piece * Catalogue::Lire(int NumPiece)
295:  {
296:     int indice;
297:     return (Rechercher(indice, NumPiece));
298:
299:  }
300:
301:  int main()
302:  {
303:     Catalogue cat;
304:     Piece * pPiece = 0;
305:     int NumPiece;
306:     int valeur;
307:     int choix = 99;
308:
309:     while (choix!= 0)
310:     {
311:        cout << "(0)Quitter (1)Auto (2)Avion - Votre choix : ";
312:        cin >> choix;
313:
314:        if (choix != 0)
315:        {
316:           cout << "Entrez un numéro de pièce : ";
317:           cin >>  NumPiece;
318:
319:           if (choix == 1)
320:           {
321:              cout << "Millésime ? ";
322:              cin >> valeur;
```

```
323:              pPiece = new PieceAuto(valeur, NumPiece);
324:           }
325:           else
326:           {
327:              cout << "Référence moteur ?: ";
328:              cin >> valeur;
329:              pPiece = new PieceAvion(valeur, NumPiece);
330:           }
331:           cat.Inserer(pPiece);
332:        }
333:     }
334:     cat.AfficherTout();
335:     return 0;
336:  }
```

Ce programme produit le résultat suivant :

```
(0)Quitter (1)Auto (2)Avion - Votre choix : 1
Entrez un numéro de pièce : 1234
Millésime ? 94
(0)Quitter (1)Auto (2)Avion - Votre choix : 1
Entrez un numéro de pièce : 4434
Millésime ? 93
(0)Quitter (1)Auto (2)Avion - Votre choix : 1
Entrez un numéro de pièce : 1234
Millésime ? 94
1234 était la première entrée. Refusée !
(0)Quitter (1)Auto (2)Avion - Votre choix : 1
Entrez un numéro de pièce : 2345
Millésime ? 93
(0)Quitter (1)Auto (2)Avion - Votre choix : 0

Numéro de pièce : 1234
Millésime : 94

Numéro de pièce : 2345
Millésime : 93

Numéro de pièce : 4434
Millésime : 93
```

Dans ce programme, nous n'avons modifié que l'interface de `Catalogue` et la fonction principale du listing précédent.

À la ligne 253, `Catalogue` est déclarée comme dérivant en privé de `ListePieces`. Désormais, `Catalogue` n'a donc plus besoin d'un membre `ListePieces`.

La fonction `AfficherTout()` de `Catalogue` (ligne 160) appelle la fonction `Traiter()` de `ListePieces` en passant le pointeur approprié à la fonction membre de la classe `Piece`. `AfficherTout()` agit comme l'interface publique de `Traiter()` en fournissant les informations adéquates mais en empêchant les classes clientes d'appeler directement `Traiter()`. Bien que `ListePieces` autorise le passage d'autres fonctions en paramètre de `Traiter()`, `Catalogue` ne le fait pas.

La fonction `Inserer()` a également été modifiée (lignes 164 à 284). À la ligne 269, `Trouver()` est désormais appelée directement, car elle a été héritée de la classe de base. Pour éviter un appel récursif sans fin de la fonction `Inserer()` sur elle-même, l'appel de cette fonction doit être pleinement qualifié (ligne 271).

En résumé, les méthodes de `Catalogue` peuvent appeler directement celles de `ListesPieces`. La seule exception concerne les cas où `Catalogue` a redéfinit une méthode et qu'il faut appeler la version de `ListesPieces` : en ce cas, cet appel doit être pleinement qualifié.

L'héritage privé permet à `Catalogue` d'hériter de ce qu'elle peut utiliser, tout en permettant de contrôler l'accès à `Insert()` et aux autres méthodes auxquelles les classes clientes ne doivent pas avoir directement accès.

Faire	*Ne pas faire*
● Utiliser l'héritage public lorsque l'objet dérivé est une sorte d'objet de la classe de base. ● Utiliser l'agrégation pour déléguer des fonctionnalités à une autre classe et que vous n'avez pas besoin d'accéder à ses membres protégés. ● Utiliser l'héritage privé lorsque vous devez implémenter une classe en termes d'une autre et que vous devez avoir accès aux membres protégés de la classe de base.	● Utiliser l'héritage privé lorsque vous devez utiliser plusieurs objets de la classe de base. En ce cas, utilisez l'agrégation. Si `Catalogue` avait besoin de deux `ListePieces`, par exemple, vous ne pourriez pas utiliser l'héritage privé. ● Utiliser l'héritage public lorsque les membres de la classe de base doivent être cachés aux clients de la classe dérivée.

Classes amies

Il peut parfois être intéressant de regrouper des classes dans un ensemble. Par exemple, `NoeudPiece` et `ListePieces` étant étroitement associées, il aurait été pratique que `ListePieces` puisse lire directement le pointeur de `Piece` de `NoeudPiece` (`saPiece`).

Il n'est pas souhaitable que `saPiece` soit publique ou même protégée car il s'agit d'un détail d'implémentation de `NoeudPiece` et qu'elle doit donc rester privée. En revanche, vous voulez que `ListePieces` puisse y accéder directement.

La solution consiste à définir une classe *amie*, ce qui permettra à cette dernière d'accéder aux données et aux fonctions privées de votre classe. Ceci étend donc l'interface de votre classe pour y inclure la classe amie.

Lorsqu'une classe déclare qu'une autre classe est son amie, toutes les données et fonctions membres de la classe déclarante sont publiques pour la classe amie. Si, par exemple, `NoeudPiece` déclare que `ListePieces` est son amie, toutes les données et méthodes membres de `NoeudPiece` seront publiques pour `ListePieces`.

Il faut bien noter que l'amitié ne peut pas être transférée. Elle n'est pas transitive : les amis de mes amis *ne sont pas* toujours mes amis. L'amitié n'est pas non plus transmise par héritage : si vous êtes mon ami et que je veux partager mes secrets avec vous, cela ne signifie pas que je veuille le faire avec vos enfants.

Elle n'est pas non plus commutative : si la classe A est amie de B, B ne devient pas de ce fait une amie de A. Vous voulez peut-être me dire vos secrets, mais cela n'implique pas que je veuille vous dire les miens.

Pour déclarer une classe amie, utilisez le mot-clé `friend` :

```
class ClasseUn
{
   public:
     friend class ClasseAmie;
     . . .
```

Dans cet exemple, `ClasseUn` a déclaré `ClasseAmie` comme son amie. Cette dernière a donc un accès total aux membres de `ClasseUn`.

Dans le Listing 16.6, `ListePieces` devient une classe amie de `NoeudPiece`, mais la réciproque n'est pas vraie.

Listing 16.7 : Implémentation d'une classe amie

```
 0:  //Listing 16.7 Implémentation d'une classe amie
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  // *************** Pièce ************
 6:
 7:  // Classe de base abstraite de pièces
 8:  class Piece
 9:  {
10:  public:
11:     Piece():saNumPiece(1) {}
```

```
12:     Piece(int NumPiece):
13:     saNumPiece(NumPiece){}
14:     virtual ~Piece(){}
15:     int GetNumPiece() const
16:        { return saNumPiece; }
17:     virtual void Afficher() const = 0;
18:  private:
19:     int saNumPiece;
20:  };
21:
22:  // implémentation d'une fonction virtuelle pure
23:  // pour chaîner les classes dérivées
24:  void Piece::Afficher() const
25:  {
26:     cout << "\nNuméro de pièce : ";
27:     cout << saNumPiece << endl;
28:  }
29:
30:  // *************** Pièce Auto ************
31:
32:  class PieceAuto : public Piece
33:  {
34:  public:
35:     PieceAuto():sonAnneeAuto(97){}
36:     PieceAuto(int annee, int NumPiece);
37:     virtual void Afficher() const
38:     {
39:        Piece::Afficher();
40:        cout << "Millésime : ";
41:        cout << sonAnneeAuto << endl;
42:     }
43:  private:
44:     int sonAnneeAuto;
45:  };
46:
47:  PieceAuto::PieceAuto(int annee, int NumPiece):
48:     sonAnneeAuto(annee),
49:     Piece(NumPiece)
50:  {}
51:
52:
53:  // *************** Pièce Avion************
54:
55:  class PieceAvion : public Piece
56:  {
57:  public:
58:     PieceAvion():saRefMoteur(1){};
```

```
59:     PieceAvion(int RefMoteur, int NumPiece);
60:     virtual void Afficher() const
61:     {
62:        Piece::Afficher();
63:        cout << "Référence moteur : ";
64:        cout << saRefMoteur << endl;
65:     }
66:  private:
67:     int saRefMoteur;
68:  };
69:
70:  PieceAvion::PieceAvion(int RefMoteur, int NumPiece):
71:     saRefMoteur(RefMoteur),
72:     Piece(NumPiece)
73:  {}
74:
75:  // ***************** Noeud Pièce ************
76:  class NoeudPiece
77:  {
78:  public:
79:     friend class ListePieces;
80:     NoeudPiece (Piece*);
81:     ~NoeudPiece();
82:     void SetSuivant(NoeudPiece * noeud)
83:        { sonSuivant = noeud; }
84:     NoeudPiece * GetSuivant () const;
85:     Piece * GetPiece () const;
86:  private:
87:     Piece *saPiece ;
88:     NoeudPiece * sonSuivant;
89:  };
90:
91:
92:  NoeudPiece::NoeudPiece(Piece* pPiece):
93:     saPiece(pPiece),
94:     sonSuivant(0)
95:  {}
96:
97:  NoeudPiece::~NoeudPiece()
98:  {
99:     delete saPiece ;
100:    saPiece  = 0;
101:    delete sonSuivant;
102:    sonSuivant = 0;
103: }
104:
105: // Renvoie NULL si pas de NoeudPiece suivant
```

```
106:  NoeudPiece * NoeudPiece::GetSuivant () const
107:  {
108:     return sonSuivant;
109:  }
110:
111:  Piece * NoeudPiece::GetPiece () const
112:  {
113:     if (saPiece )
114:        return saPiece ;
115:     else
116:        return NULL; // erreur
117:  }
118:
119:
120:  // *************** Liste Pièces ************
121:  class ListePieces
122:  {
123:  public:
124:     ListePieces();
125:     ~ListePieces();
126:     // nécessite constructeur de copie et opérateur Egal !
127:     void    Traiter(void (Piece::*f)()const) const;
128:     Piece*  Rechercher(int & position, int NumPiece) const;
129:     Piece*  GetPremier() const;
130:     void    Inserer(Piece *);
131:     Piece*  operator[](int) const;
132:     int     GetTotal() const { return sonTotal; }
133:     static  ListePieces& GetListePiecesGlobale()
134:     {
135:        return  ListePiecesGlobale;
136:     }
137:  private:
138:     NoeudPiece * pTete;
139:     int sonTotal;
140:     static ListePieces ListePiecesGlobale;
141:  };
142:
143:  ListePieces ListePieces::ListePiecesGlobale;
144:
145:  // Listes : implémentations...
146:
147:  ListePieces::ListePieces():
148:     pTete(0),
149:     sonTotal(0)
150:  {}
151:
152:  ListePieces::~ListePieces()
```

```
153:  {
154:     delete pTete;
155:  }
156:
157:  Piece*   ListePieces::GetPremier() const
158:  {
159:     if (pTete)
160:        return pTete->saPiece ;
161:     else
162:        return NULL;  // interception d'erreur
163:  }
164:
165:  Piece * ListePieces::operator[](int indice) const
166:  {
167:     NoeudPiece* pNoeud = pTete;
168:
169:     if (!pTete)
170:        return NULL; // interception d'erreur
171:
172:     if (indice > sonTotal)
173:        return NULL; //  erreur
174:
175:     for (int i=0; i < indice; i++)
176:        pNoeud = pNoeud->sonSuivant;
177:
178:     return   pNoeud->saPiece ;
179:  }
180:
181:  Piece* ListePieces::Rechercher(int & position, int NumPiece) const
182:  {
183:     NoeudPiece * pNoeud = 0;
184:     for (pNoeud = pTete, position = 0;
185:        pNoeud!=NULL;
186:     pNoeud = pNoeud->sonSuivant, position++)
187:     {
188:        if (pNoeud->saPiece ->GetNumPiece() == NumPiece)
189:           break;
190:     }
191:     if (pNoeud == NULL)
192:        return NULL;
193:     else
194:        return pNoeud->saPiece ;
195:  }
196:
197:  void ListePieces::Traiter(void (Piece::*func)()const) const
198:  {
199:     if (!pTete)
```

```
200:        return;
201:     NoeudPiece* pNoeud = pTete;
202:     do
203:        (pNoeud->saPiece ->*func)();
204:     while (pNoeud = pNoeud->sonSuivant);
205:  }
206:
207:  void ListePieces::Inserer(Piece* pPiece)
208:  {
209:     NoeudPiece * pNoeud = new NoeudPiece(pPiece);
210:     NoeudPiece * pCourant = pTete;
211:     NoeudPiece * pSuivant = 0;
212:
213:     int Nouveau =  pPiece->GetNumPiece();
214:     int Suivant = 0;
215:     sonTotal++;
216:
217:     if (!pTete)
218:     {
219:        pTete = pNoeud;
220:        return;
221:     }
222:
223:     // si celui-ci est plus petit que Tete
224:     // il devient le nouveau noeud de Tete
225:     if (pTete->saPiece ->GetNumPiece() > Nouveau)
226:     {
227:        pNoeud->sonSuivant = pTete;
228:        pTete = pNoeud;
229:        return;
230:     }
231:
232:     for (;;)
233:     {
234:        // si pas de suivant, ajouter celui-ci
235:        if (!pCourant->sonSuivant)
236:        {
237:           pCourant->sonSuivant = pNoeud;
238:           return;
239:        }
240:
241:        // s'il vient après celui-ci et avant le suivant
242:        // l'insérer ici, sinon lire le suivant
243:        pSuivant = pCourant->sonSuivant;
244:        Suivant = pSuivant->saPiece ->GetNumPiece();
245:        if (Suivant > Nouveau)
246:        {
```

```
247:           pCourant->sonSuivant = pNoeud;
248:           pNoeud->sonSuivant = pSuivant;
249:           return;
250:        }
251:        pCourant = pSuivant;
252:     }
253:  }
254:
255:  class Catalogue : private ListePieces
256:  {
257:  public:
258:     void Inserer(Piece *);
259:     int Existe(int NumPiece);
260:     Piece * Lire(int NumPiece);
261:     operator+(const Catalogue &);
262:     void AfficherTout() { Traiter(Piece::Afficher); }
263:  private:
264:  };
265:
266:  void Catalogue::Inserer(Piece * nouvPiece )
267:  {
268:     int NumPiece = nouvPiece->GetNumPiece();
269:     int indice;
270:
271:     if (!Rechercher(indice, NumPiece))
272:        ListePieces::Inserer(nouvPiece);
273:     else
274:     {
275:        cout << NumPiece << " était la ";
276:        switch (indice)
277:        {
278:        case 0:  cout << "première "; break;
279:        case 1:  cout << "deuxième "; break;
280:        case 2:  cout << "troisième "; break;
281:        default: cout << indice + 1 << "e";
282:        }
283:        cout << "entrée. Refusée !" << endl;
284:     }
285:  }
286:
287:  int Catalogue::Existe(int NumPiece)
288:  {
289:     int indice;
290:     Rechercher(indice, NumPiece);
291:     return indice;
292:  }
293:
```

```
294:  Piece * Catalogue::Lire(int NumPiece)
295:  {
296:     int indice;
297:     return (Rechercher(indice, NumPiece));
298:  }
299:
300:  int main()
301:  {
302:     Catalogue cat;
303:     Piece * pPiece = 0;
304:     int NumPiece;
305:     int valeur;
306:     int choix = 99;
307:
308:     while (choix!= 0)
309:     {
310:        cout << "(0)Quitter (1)Auto (2)Avion - Votre choix : ";
311:        cin >> choix;
312:
313:        if (choix != 0)
314:        {
315:           cout << "Entrez un numéro de pièce : ";
316:           cin >>  NumPiece;
317:
318:           if (choix == 1)
319:           {
320:              cout << "Millésime ? ";
321:              cin >> valeur;
322:              pPiece = new PieceAuto(valeur, NumPiece);
323:           }
324:           else
325:           {
326:              cout << "Référence moteur ?: ";
327:              cin >> valeur;
328:              pPiece = new PieceAvion(valeur, NumPiece);
329:           }
330:           cat.Inserer(pPiece);
331:        }
332:     }
333:     cat.AfficherTout();
334:     return 0;
335:  }
```

Ce programme produit le résultat suivant :

```
(0)Quitter (1)Auto (2)Avion - Votre choix : 1
Entrez un numéro de pièce : 1234
Millésime ? 94
```

```
(0)Quitter (1)Auto (2)Avion - Votre choix : 1
Entrez un numéro de pièce : 4434
Millésime ? 93
(0)Quitter (1)Auto (2)Avion - Votre choix : 1
Entrez un numéro de pièce : 1234
Millésime ? 94
1234 était la première entrée. Refusée !
(0)Quitter (1)Auto (2)Avion - Votre choix : 1
Entrez un numéro de pièce : 2345
Millésime ? 93
(0)Quitter (1)Auto (2)Avion - Votre choix : 0

Numéro de pièce : 1234
Millésime : 94

Numéro de pièce : 2345
Millésime : 93

Numéro de pièce : 4434
Millésime : 93
```

À la ligne 79, la classe `ListePieces` est déclarée amie de la classe `NoeudPiece`.

La déclaration de la relation d'amitié figure dans la section publique, mais elle pourrait être placée n'importe où dans la déclaration de classe sans modifier pour autant la signification de l'instruction. Grâce à cette amitié, tous les membres et méthodes privés de `NoeudPiece` seront disponibles depuis n'importe quelle fonction membre de la classe `ListePieces`.

À la ligne 157, l'implémentation de la fonction membre `GetPremier()` reflète cette modification. Au lieu de renvoyer `pTete->GetPiece()` , elle peut désormais se contenter de renvoyer le membre privé `pTete->saPiece`. De même, la fonction `Inserer()` écrit désormais `pNoeud->sonSuivant = pTete`, au lieu de `pNoeud->SetSuivant(pTete)`.

Il s'agit, bien sûr, de modifications triviales qui ne justifient pas que l'on ait besoin que `ListePieces` soit amie de `NoeudPiece`, mais l'objectif, ici, est simplement d'illustrer le fonctionnement du mot-clé `friend`.

Les déclarations de classes amies doivent être utilisées avec la plus grande prudence. Si deux classes sont étroitement liées et que l'une a fréquemment besoin des données de l'autre, établir une relation d'amitié constitue une bonne solution. Mais il faut l'utiliser avec parcimonie ; il est parfois tout aussi simple de se servir de méthodes d'accès publiques. En outre, cela vous évite de devoir recompiler les deux classes lorsque l'une d'elle est modifiée.



Vous entendrez souvent les débutants en C++ se plaindre que les déclarations d'amitié sapent l'encapsulation des données, qui est si importante en programmation orientée objet. Ce n'est pas nécessairement vrai : une déclaration d'amitié permet d'intégrer la partie amie à l'interface de la classe, ce qui ne pertube pas forcément l'encapsulation. L'utilisation d'une amie, par contre, implique de s'engager à maintenir en parallèle les deux classes, ce qui peut réduire la modularité du code.

Mot-clé `friend`

Pour déclarer une classe amie d'une autre classe, il suffit de faire précéder son nom du mot-clé `friend` dans la classe qui donne les droits d'accès. En d'autres termes, je peux déclarer que vous êtes mon ami, mais vous ne pouvez pas le décider vous-même.

Exemple

```
class NoeudPiece{
public:
friend class ListePieces;
// ListePieces est devenue amie de NoeudPiece
};
```

Fonctions amies

Nous venons de voir qu'une déclaration de classe amie donne un accès total. Parfois, on ne ne souhaite pas accorder ce niveau d'accès à toute une classe, mais uniquement à une ou deux de ses méthodes. Pour ce faire, on peut déclarer amies les méthodes membres de l'autre classe au lieu de déclarer toute celle-ci comme amie. En fait, n'importe quelle fonction, qu'elle soit membre d'une classe ou non, peut être déclarée amie.

Fonctions amies et surcharge d'opérateur

Dans le Listing 16.1, la classe `String` redéfinissait l'opérateur +. Elle incluait également un constructeur qui prenait en paramètre un pointeur constant sur un caractère afin de pouvoir créer des objets `String` à partir de chaînes de type C. Ces deux méthodes permettent de créer un objet `String` et de le concaténer avec une chaîne de type C.



Une chaîne de style C est un tableau de caractères terminé par le caractère nul. Exemple :

```
char uneChaine[] = "Bonjour".
```

Cependant, vous ne pouvez pas créer une chaîne C pour lui ajouter ensuite un objet `String`, comme ici :

```
char uneChaineC[] = "Bonjour";
String unString(" les amis");
String unString2 = uneChaineC + unString; // erreur !
```

Les chaînes C ne disposent pas d'un opérateur + surchargé. Comme vous l'avez vu au Chapitre 10, l'expression `uneChaineC + unString;` revient à écrire `uneChaineC.operator+(unString)`. Comme vous ne pouvez pas appeler la méthode `operator+()` sur une chaîne C, cela produira une erreur de compilation.

Pour résoudre ce problème, vous pouvez déclarer une fonction amie dans `String`, qui surcharge `operator+()` pour qu'elle prenne deux paramètres de type `String`. La chaîne C sera alors convertie en objet `String` à l'aide du constructeur approprié, puis `operator+()` sera appelée avec les deux objets. Pour plus de détails, examinez le Listing 16.8.

Listing 16.8 : Un opérateur + amical

```
 0:  //Listing 16.8 - opérateurs amis
 1:
 2:  #include <iostream>
 3:  #include <String.h>
 4:  using namespace std;
 5:
 6:  // Classe String rudimentaire
 7:  class String
 8:  {
 9:  public:
10:     // constructeurs
11:     String();
12:     String(const char *const);
13:     String(const String &);
14:     ~String();
15:
16:     // opérateurs surcharges
17:     char & operator[](int indice);
18:     char operator[](int indice) const;
19:     String operator+(const String&);
20:     friend String operator+(const String&, const String&);
21:     void operator+=(const String&);
22:     String & operator= (const String &);
23:
24:     // méthodes d'accès générales
25:     int GetLongueur()const { return saLongueur; }
26:     const char * GetString() const { return saChaine; }
```

```
27:
28:  private:
29:     String (int);         // constructeur privé
30:     char * saChaine;
31:     unsigned short saLongueur;
32:  };
33:
34:  // le constructeur par défaut crée des chaînes de 0 octet
35:  String::String()
36:  {
37:     saChaine = new char[1];
38:     saChaine[0] = '\0';
39:     saLongueur=0;
40:     // cout << "\tConstructeur par défaut de String" << endl;
41:     // CptConstructeur++;
42:  }
43:
44:  // constructeur (utilitaire) privé, utilisé seulement par
45:  // les méthodes de la classe pour créer une nouvelle chaîne
46:  // de la taille requise, contenant des caractères nuls.
47:  String::String(int longueur)
48:  {
49:     saChaine = new char[longueur+1];
50:     for (int i = 0; i <= longueur; i++)
51:        saChaine[i] = '\0';
52:     saLongueur = longueur;
53:     // cout << "\tConstructeur de String(int)" << endl;
54:     // CptConstructeur++;
55:  }
56:
57:  // Convertit un tableau de caractères en String
58:  String::String(const char * const chaineC)
59:  {
60:     saLongueur = strlen(chaineC);
61:     saChaine = new char[saLongueur+1];
62:     for (int i = 0; i < saLongueur; i++)
63:        saChaine[i] = chaineC[i];
64:     saChaine[saLongueur]='\0';
65:     // cout << "\tConstructeur de String(char*)" << endl;
66:     // CptConstructeur++;
67:  }
68:
69:  // constructeur de copie
70:  String::String (const String & rhs)
71:  {
72:     saLongueur=rhs.GetLongueur();
73:     saChaine = new char[saLongueur+1];
```

```
74:     for (int i = 0; i < saLongueur;i++)
75:        saChaine[i] = rhs[i];
76:     saChaine[saLongueur] = '\0';
77:     // cout << "\tConstructeur de String(String&)" << endl;
78:     // CptConstructeur++;
79:  }
80:
81:  // destructeur, libère la mémoire
82:  String::~String ()
83:  {
84:     delete [] saChaine;
85:     saLongueur = 0;
86:     // cout << "\tDestructeur de String" << endl;
87:  }
88:
89:  // opérateur d'affectation, libère la mémoire existante
90:  // puis copie la chaîne et la taille
91:  String& String::operator=(const String & rhs)
92:  {
93:     if (this == &rhs)
94:        return *this;
95:     delete [] saChaine;
96:     saLongueur=rhs.GetLongueur();
97:     saChaine = new char[saLongueur+1];
98:     for (int i = 0; i < saLongueur; i++)
99:        saChaine[i] = rhs[i];
100:    saChaine[saLongueur] = '\0';
101:    return *this;
102:    // cout << "\toperator= de String" << endl;
103: }
104:
105: // opérateur d'indexation non constant, renvoie
106: // une référence sur caractère pour qu'il puisse
107: // être modifié !
108: char & String::operator[](int indice)
109: {
110:    if (indice > saLongueur)
111:       return saChaine[saLongueur-1];
112:    else
113:       return saChaine[indice];
114: }
115:
116: // opérateur d'indexation constant pour utilisation
117: // sur des objets const (voir constructeur de copie !)
118: char String::operator[](int indice) const
119: {
120:    if (indice > saLongueur)
```

```
121:        return saChaine[saLongueur-1];
122:     else
123:        return saChaine[indice];
124:  }
125:
126:  // crée une nouvelle chaîne en ajoutant
127:  // la chaîne actuelle à rhs
128:  String String::operator+(const String& rhs)
129:  {
130:     int longueurTotale = saLongueur + rhs.GetLongueur();
131:     String temp(longueurTotale);
132:     int i, j;
133:     for (i = 0; i < saLongueur; i++)
134:        temp[i] = saChaine[i];
135:     for (j=0, i=saLongueur; j < rhs.GetLongueur(); j++, i++)
136:        temp[i] = rhs[j];
137:     temp[longueurTotale]='\0';
138:     return temp;
139:  }
140:
141:  // crée une nouvelle chaîne en ajoutant
142:  // une chaîne à une autre
143:  String operator+(const String& lhs, const String& rhs)
144:  {
145:     int longueurTotale = lhs.GetLongueur()+rhs.GetLongueur();
146:     String temp(longueurTotale);
147:     int i, j;
148:     for (i = 0; i < lhs.GetLongueur(); i++)
149:        temp[i] = lhs[i];
150:     for (j=0, i=lhs.GetLongueur(); j<rhs.GetLongueur()
150a:                                  ; j++, i++)
151:        temp[i] = rhs[j];
152:     temp[longueurTotale]='\0';
153:     return temp;
154:  }
155:
156:  int main()
157:  {
158:     String s1("String Une ");
159:     String s2("String Deux ");
160:     char *c1 = { "Chaine C Une " } ;
161:     String s3;
162:     String s4;
163:     String s5;
164:
165:     cout << "s1 : " << s1.GetString() << endl;
166:     cout << "s2 : " << s2.GetString() << endl;
```

```
167:     cout << "c1 : " << c1 << endl;
168:     s3 = s1 + s2;
169:     cout << "s3 : " << s3.GetString() << endl;
170:     s4 = s1 + c1;
171:     cout << "s4 : " << s4.GetString() << endl;
172:     s5 = c1 + s2;
173:     cout << "s5 : " << s5.GetString() << endl;
174:     return 0;
175: }
```

Ce programme produit le résultat suivant :

```
s1 : String Une
s2 : String Deux
c1 : Chaine C Une
s3 : String Une String Deux
s4 : String Une Chaine C Une
s5 : Chaine C Une String Deux
```

Hormis l'implémentation de `operator+`, les méthodes de la classe `String` n'ont pas été modifiées par rapport au Listing 16.1. La ligne 20 surcharge `operator+` pour qu'elle prenne en paramètre deux références constantes de `String` et qu'elle renvoie un `String`. Cette fonction est déclarée amie.

Vous remarquerez que cette version de `operator+` n'est membre d'aucune classe. Cette fonction est déclarée dans la classe `String` uniquement pour qu'elle puisse devenir son amie. En outre, comme elle est déclarée, vous n'avez pas besoin d'un prototype de fonction.

L'implémentation de cette version de `operator+` apparaît de la ligne 143 à la ligne 154. Cette fonction ressemble à la méthode `operator+` précédente, sauf qu'elle attend deux `String` en paramètres et qu'elle y accède grâce à leurs méthodes d'accès publiques.

La ligne 172 du programme de test montre que l'on peut maintenant appeler cet opérateur sur une chaîne C !

Les fonctions amies

On déclare une fonction amie à l'aide du mot-clé `friend` suivi de la spécification complète de la fonction. Le fait de déclarer une fonction amie ne lui donne pas accès à votre pointeur `this`, mais lui permet d'accéder à l'ensemble de vos données et de vos fonctions membres privées et protégées.

Exemple

```
class NoeudPiece
{
   // une fonction membre d'une autre
```

```
    // classe est déclarée amie
    friend void ListePieces::Inserer(Piece *);
    // une fonction globale est déclarée amie
    friend int Fonction();
    //...
};
```

Surcharge de *operator<<()*

La classe `String` est désormais prête à utiliser `cout` comme n'importe quel autre type standard. Jusqu'à présent, pour afficher un objet `String`, il fallait écrire une instruction comme celle-ci :

```
cout << uneString.GetString();
```

Alors que vous auriez préféré écrire ceci :

```
cout << uneString;
```

Pour cela, vous devez redéfinir `operator<<()`. Au Chapitre 17, vous apprendrez à gérer les flux d'entrée-sortie mais, pour le moment, nous allons voir avec le Listing 16.9 comment surcharger `operator<<()` à l'aide d'une fonction amie.

Listing 16.9 : Surcharge de l'opérateur *<<()*

```
 0:  // Listing 16.9 Surcharge de operator<<()
 1:
 2:  #include <iostream>
 3:  #include <string.h>
 4:  using namespace std;
 5:
 6:  class String
 7:  {
 8:  public:
 9:      // constructeurs
10:     String();
11:     String(const char *const);
12:     String(const String &);
13:     ~String();
14:
15:     // opérateurs surchargés
16:     char & operator[](int indice);
17:     char operator[](int indice) const;
18:     String operator+(const String&);
```

```
19:     void operator+=(const String&);
20:     String & operator= (const String &);
21:     friend ostream& operator<<
22:        ( ostream& leFlux, String& leString);
23:     // méthodes générales d'accès
24:     int GetLongueur()const { return saLongueur; }
25:     const char * GetString() const { return saChaine; }
26:
27:  private:
28:     String (int);         // constructeur privé
29:     char * saChaine;
30:     unsigned short saLongueur;
31:  };
32:
33:
34:  // le constructeur par défaut crée des chaînes de 0 octet
35:  String::String()
36:  {
37:     saChaine = new char[1];
38:     saChaine[0] = '\0';
39:     saLongueur=0;
40:     // cout << "\tConstructeur par défaut de String" << endl;
41:     // CptConstructeur++;
42:  }
43:
44:  // constructeur (utilitaire) privé, utilisé seulement par
45:  // les méthodes de la classe pour créer une nouvelle
46:  // chaîne de la taille requise, contenant des caractères nuls.
47:  String::String(int longueur)
48:  {
49:     saChaine = new char[longueur+1];
50:     for (int i = 0; i<=longueur; i++)
51:        saChaine[i] = '\0';
52:     saLongueur=longueur;
53:     // cout << "\tConstructeur de String(int)" << endl;
54:     // CptConstructeur++;
55:  }
56:
57:  // Convertit un tableau de caractères en String
58:  String::String(const char * const chaineC)
59:  {
60:     saLongueur = strlen(chaineC);
61:     saChaine = new char[saLongueur+1];
62:     for (int i = 0; i < saLongueur; i++)
63:        saChaine[i] = chaineC[i];
64:     saChaine[saLongueur]='\0';
65:     // cout << "\tConstructeur de String(char*)" << endl;
```

```
66:     // CptConstructeur++;
67:  }
68:
69:  // constructeur de copie
70:  String::String (const String & rhs)
71:  {
72:     saLongueur = rhs.GetLongueur();
73:     saChaine = new char[saLongueur+1];
74:     for (int i = 0; i < saLongueur; i++)
75:        saChaine[i] = rhs[i];
76:     saChaine[saLongueur] = '\0';
77:     // cout << "\tConstructeur de String(String&)" << endl;
78:     // CptConstructeur++;
79:  }
80:
81:  // destructeur, libère la mémoire
82:  String::~String ()
83:  {
84:     delete [] saChaine;
85:     saLongueur = 0;
86:     // cout << "\tDestructeur de String" << endl;
87:  }
88:
89:  // opérateur d'affectation, libère la mémoire existante
90:  // puis copie la chaîne et la taille
91:  String& String::operator=(const String & rhs)
92:  {
93:     if (this == &rhs)
94:        return *this;
95:     delete [] saChaine;
96:     saLongueur=rhs.GetLongueur();
97:     saChaine = new char[saLongueur+1];
98:     for (int i = 0; i < saLongueur; i++)
99:        saChaine[i] = rhs[i];
100:    saChaine[saLongueur] = '\0';
101:    return *this;
102:    // cout << "\tOperator= de String" << endl;
103: }
104:
105: // opérateur d'indexation non constant, renvoie
106: // une référence sur caractère pour qu'il puisse
107: // être modifié !
108: char & String::operator[](int indice)
109: {
110:    if (indice > saLongueur)
111:       return saChaine[saLongueur-1];
112:    else
```

```
113:        return saChaine[indice];
114:  }
115:
116:  // opérateur d'indexation constant pour utilisation
117:  // sur les objets const (voir constructeur de copie !)
118:  char String::operator[](int indice) const
119:  {
120:     if (indice > saLongueur)
121:        return saChaine[saLongueur-1];
122:     else
123:        return saChaine[indice];
124:  }
125:
126:  // crée une nouvelle chaîne en ajoutant
127:  // la chaîne actuelle à rhs
128:  String String::operator+(const String& rhs)
129:  {
130:     int  longueurTotale = saLongueur + rhs.GetLongueur();
131:     String temp(longueurTotale);
132:     int i, j;
133:     for (i = 0; i < saLongueur; i++)
134:        temp[i] = saChaine[i];
135:     for (j = 0; j < rhs.GetLongueur(); j++, i++)
136:        temp[i] = rhs[j];
137:     temp[longueurTotale]='\0';
138:     return temp;
139:  }
140:
141:  // modifie la chaîne courante, ne renvoie rien
142:  void String::operator+=(const String& rhs)
143:  {
144:     unsigned short rhsLongueur = rhs.GetLongueur();
145:     unsigned short longueurTotale = saLongueur + rhsLongueur;
146:     String  temp(longueurTotale);
147:     int i, j;
148:     for (i = 0; i < saLongueur; i++)
149:        temp[i] = saChaine[i];
150:     for (j = 0, i = 0; j < rhs.GetLongueur(); j++, i++)
151:        temp[i] = rhs[i - saLongueur];
152:     temp[longueurTotale]='\0';
153:     *this = temp;
154:  }
155:
156:  // int String::CptConstructeur =
157:  ostream& operator<<(ostream& leFlux,String& leString)
158:  {
159:     leFlux << leString.saChaine;
```

```
160:     return leFlux;
161:  }
162:
163:  int main()
164:  {
165:     String unString("Bonjour.");
166:     cout << unString;
167:     return 0;
168:  }
```

Ce programme produit le résultat suivant :

```
Bonjour.
```

Les lignes 21 et 22 déclarent `operator<<()` comme une fonction amie prenant en paramètre une référence à un `ostream` et une référence à `String`, et renvoyant une référence à un `ostream`. Il ne s'agit pas d'une fonction membre de `String`. Elle renvoie une référence à un `ostream` afin de vous permettre d'enchaîner ses appels, comme ici :

```
cout << "Age : " << sonAge << "an(s).";
```

Cette fonction amie est implémentée de la ligne 157 à la ligne 161. En fait, vous pouvez constater qu'elle se borne à cacher le transfert de l'objet `String` vers l'objet `ostream`. Le Chapitre 17 reviendra en détail sur la surcharge de `operator<<()` et `operator>>()`..

Questions-réponses

Q Pourquoi est-il si important de faire la distinction entre les relations *est-un*, *a-un* et *implémenté en termes de* ?

R L'objectif de C++ est de vous permettre de réaliser des programmes orientés objet clairs et optimisés. Distinguer clairement ces types de relations permet de s'assurer que la conception correspond bien à la réalité que l'on souhaite modéliser. En outre, une conception propre se reflètera dans un code bien contruit.

Q Qu'est-ce que la composition ?

R C'est un synonyme de l'agrégation.

Q Pourquoi préférer l'agrégation à l'héritage privé ?

R De nos jours, le développeur cherche à simplifier les choses pour mieux les maîtriser. Plus vous pouvez utiliser les objets comme des boîtes noires, moins vous aurez de détails à vous soucier et mieux vous pourrez gérer la complexité. L'agrégation est une technique d'encapsulation plus fiable que l'héritage privé, car ce dernier ne masque pas l'implémentation des objets de la classe. Dans une certaine mesure, cela vaut aussi

pour l'héritage public traditionnel, qui est parfois utilisé alors qu'une agrégation serait une meilleure solution.

Q Pourquoi toutes les classes ne seraient-elles pas amies ?

R Déclarer une classe amie expose les détails d'implémentation et réduit l'encapsulation. L'idée consiste à masquer le plus possible les détails de chaque classe aux autres classes.

Q Si une fonction amie est surchargée, dois-je déclarer chaque version de cette fonction comme amie ?

R Oui. Si vous surchargez une fonction amie d'une classe, vous devez déclarer amies toutes les variantes de cette fonction auxquelles vous voulez donner cet accès.

Testez vos connaissances

1. Comment établit-on une relation *est-un* ?
2. Comment établit-on une relation *a-un* ?
3. Quelle est la différence entre agrégation et délégation ?
4. Quelle est la différence entre délégation et *implémentation en termes de* ?
5. Qu'est-ce qu'une fonction amie ?
6. Qu'est-ce qu'une classe amie ?
7. Si A est ami de B, la réciproque est-elle vraie ?
8. Si A est ami de B et que C dérive de A, C est-il ami de B ?
9. Si A est ami de B et B ami de C, A est-il ami de C ?
10. Où doit apparaître la déclaration d'une fonction amie ?

Exercices

1. Déclarez la classe `Animal` contenant une donnée membre de type `String`.
2. Déclarez la classe `TableauBorne`, qui est un tableau.
3. Déclarez la classe `Ensemble` en termes de tableau.
4. Modifiez le Listing 16.1, de sorte que la classe `String` contienne `operator>>()`.
5. CHERCHEZ L'ERREUR dans ce programme :

```
0:    //Cherchez l'erreur
1:    #include <iostream>
```

```
 2:    using namespace std;
 3:    class Animal;
 4:
 5:    void SetVal(Animal&, int);
 6:
 7:    class Animal
 8:    {
 9:    public:
10:       int GetPoids() const {return sonPoids; }
11:       int GetAge() const {return sonAge; }
12:    private
13:       int sonPoids;
14:       int sonAge;
15:    };
16:
17:    void SetVal(Animal& unAnimal, int lePoids)
18:    {
19:       friend class Animal;
20:       unAnimal.sonPoids = lePoids;
21:    }
22:
23:    int main()
24:    {
25:       Animal medor;
26:       SetVal(medor, 5);
27:    return 0;
28:    }
```

6. Corrigez les erreurs du listing ci-dessus pour qu'il puisse se compiler.
7. CHERCHEZ L'ERREUR dans ce programme :

```
 0:    //Cherchez l'erreur
 1:    #include  <iostream>
 2:    using namespace std;
 3:    class Animal;
 4:
 5:    void SetVal(Animal&, int);
 6:    void SetVal(Animal&, int, int);
 7:
 8:    class Animal
 9:    {
10:    friend void SetVal(Animal& ,int);  // modification !
11:    private:
12:       int sonPoids;
13:       int sonAge;
14:    };
15:
```

```
16:    void SetVal(Animal& unAnimal, int lePoids)
17:    {
18:         unAnimal.sonPoids = lePoids;
19:    }
20:
21:    void SetVal(animal& unAnimal, int lePoids, int unAge)
22:    {
23:         unAnimal.sonPoids = lePoids;
24:         unAnimal.sonAge = unAge;
25:    }
26:
27:    int main()
28:    {
29:         Animal medor;
30:         SetVal(medor, 5);
31:         SetVal(medor, 7, 9);
32:         return 0;
33:    }
```

8. Corrigez les erreurs du listing ci-dessus pour qu'il puisse se compiler.



17

Les flux

Au sommaire de ce chapitre

- Les flux et la façon de les utiliser
- Gérer les entrées/sorties avec les flux
- Écrire ou lire des fichiers avec les flux

Jusqu'à présent, nous avons utilisé `cout` pour écrire à l'écran et `cin` pour lire les données entrées au clavier, sans connaître le détail de leur mode de fonctionnement.

Présentation des flux

Le langage C++ ne définit pas la manière dont les données sont écrites à l'écran ou dans un fichier, ni la manière dont elles sont lues. Ce sont cependant des parties importantes lorsque l'on programme en C++ et la bibliothèque standard inclut donc la bibliothèque `iostream` afin de faciliter les entrées/sorties (E/S).

En séparant les E/S du langage et en les déplaçant dans des bibliothèques, il est plus facile d'obtenir un langage portable, indépendant de la plate-forme. Cela signifie que vous pouvez, en théorie, écrire des programmes C++ sur un PC, puis les recompiler et les exécuter sur une station Sun ou prendre un code créé à l'aide de l'environnement Visual C++ sous Windows et le recompiler sous Linux. Il suffit pour cela que le constructeur fournisse les bonnes bibliothèques.



Une bibliothèque est un ensemble de fichiers objet (`.obj` ou `.o`) qui peuvent être liés avec votre programme pour apporter une fonctionnalité supplémentaire. C'est donc la forme la plus basique de réutilisation du code et elle existe depuis que les anciens programmeurs ont commencé à graver des 0 et des 1 sur les murs des cavernes.

Les flux sont aujourd'hui moins importants pour la programmation C++, sauf peut-être pour la lecture des fichiers plats. Les programmes C++ utilisent désormais les bibliothèques d'interface graphique du système d'exploitation ou celles du fabricant du compilateur pour travailler avec l'écran, les fichiers et l'utilisateur : les bibliothèques Windows et les bibliothèques X Window fournissent en effet des abstractions pour les environnements Windows et X Window. Ces bibliothèques étant spécifiques au système d'exploitation et ne faisant pas partie de la norme C++, elles ne seront pas traitées dans cet ouvrage.

Les flux faisant partie de cette norme, nous allons les présenter ici. En outre, il est bon de comprendre leur fonctionnement pour appréhender le mécanisme des entrées et des sorties. Vous devriez toutefois rapidement passer à la bibliothèque GUI de votre fournisseur ou de votre système d'exploitation.

Encapsulation des flux de données

L'entrée et la sortie de texte sont réalisées à l'aide des classes `iostream`. Celles-ci considèrent le flot des données comme un *flux* d'octets, où un octet est transmis après l'autre. Ces données peuvent être envoyées du programme vers l'écran de la console ou vers un fichier, ou inversement, du clavier ou d'un fichier disque vers les variables du programme.

L'une des principales fonctions des flux est d'encapsuler le problème du transit des données, quel que soit sa direction. Après avoir créé un flux, le programme travaille avec lui et c'est lui qui règle les détails. Ce principe fondamental est illustré à la Figure 17.1.

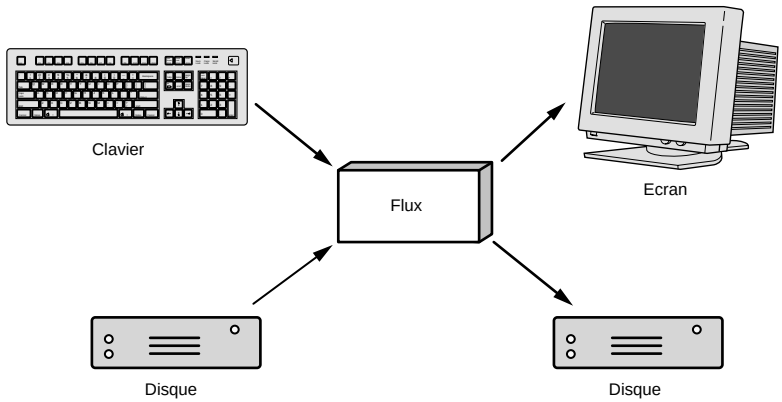


Figure 17.1
Encapsulation via *les flux.*

Utilisation des tampons

Les E/S disques (et, dans une moindre mesure, les sorties sur l'écran) sont relativement lentes et "freinent" souvent l'exécution des programmes. Pour résoudre ce problème, les flux utilisent des *tampons*. Les données sont bien écrites dans le flux, mais elles ne sont pas immédiatement transférées vers le disque. Elles sont placées dans un tampon qui se remplit à fur et à mesure et qui est transféré en une seule fois sur le disque lorsqu'il est plein.

Imaginez un robinet déversant de l'eau dans un réservoir qui se remplit progressivement, sans que l'eau ne s'écoule par le robinet du bas, comme dans la Figure 17.2.

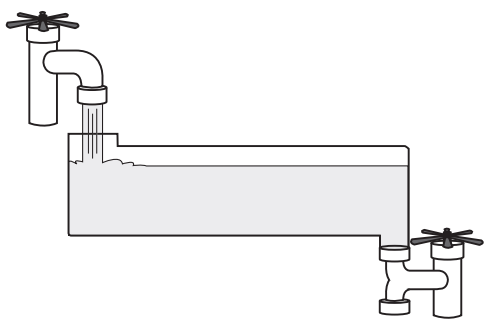


Figure 17.2
Remplissage du tampon.

Lorsque l'eau (les données) atteignent le haut du réservoir, une valve s'ouvre et toute l'eau s'écoule d'un seul coup (voir Figure 17.3).

Lorsque le tampon est vide, la valve d'écoulement se ferme, celle du haut s'ouvre à nouveau et de l'eau entre à nouveau dans le réservoir, comme dans la Figure 17.4.

Parfois, vous devez vider le réservoir même s'il n'est pas plein (voir Figure 17.5).

Figure 17.3
Vidage du tampon.

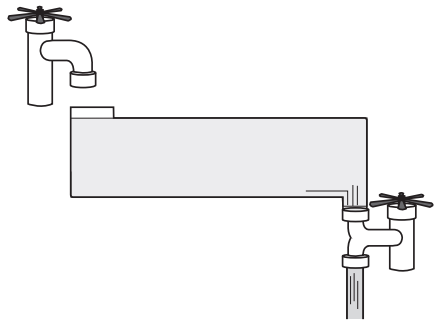
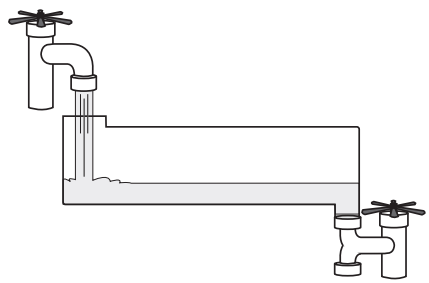
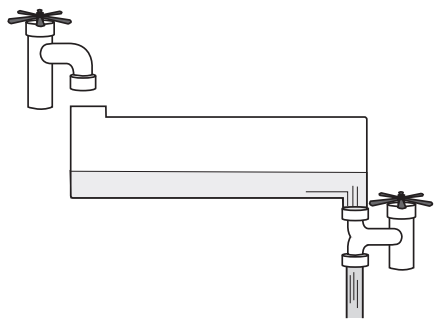


Figure 17.4
Nouveau remplissage du tampon.



C'est ce que l'on désigne par l'expression "purger le tampon".

Figure 17.5
Purge du tampon.



L'un des risques de l'utilisation des tampons est que le programme se plante alors que des données sont toujours dans les tampons. En ce cas, il est possible de perdre des données.

Flux et tampons

Comme on pouvait s'y attendre, C++ implémente les flux et les tampons selon une approche orientée objet, grâce à un certain nombre de classes et d'objets :

- La classe `streambuf` gère le tampon, ses fonctions membres permettent de remplir, vider, purger et manipuler le tampon.

- La classe `ios` est la classe de base des classes de flux d'entrée et de sortie. Elle possède un objet `streambuf` comme variable membre.
- Les classes `istream` et `ostream` dérivent de la classe `ios` et spécialisent, respectivement, le comportement des flux d'entrée et de sortie.
- La classe `iostream` dérive des classes `istream` et `ostream` ; elle fournit les méthodes d'entrée et de sortie pour l'affichage à l'écran.
- Les classes `fstream` gèrent les entrées/sorties fichiers.

Toutes ces classes seront présentées dans ce chapitre.

Les objets E/S standard

Quand un programme C++ inclut les classes `iostream`, quatre objets sont créés et initialisés :



La bibliothèque de classe `iostream` est ajoutée automatiquement par le compilateur à votre programme. La seule chose qu'il vous reste à faire pour utiliser ces fonctions consiste à placer la directive `#include` appropriée au début de votre code, comme nous l'avons déjà fait à de nombreuses reprises :

```
#include <iostream>
```

- `cin` gère les entrées en provenance de l'entrée standard : le clavier.
- `cout` gère les sorties vers la sortie standard : l'écran.
- `cerr` gère les sorties vers la sortie standard des erreurs : l'écran. `cerr` n'utilisant pas de tampon, les données qui lui sont envoyées sont donc immédiatement affichées à l'écran.
- `clog` gère les messages d'erreur qui sont envoyés vers la sortie standard des erreurs : l'écran. Ces messages, qui transitent par un tampon, sont souvent "redirigés" vers un fichier journal, comme on l'explique dans la section suivante.

Redirection des flux standard

Chacun des flux standard (entrée, sortie et erreur) peut être *redirigé* vers un autre périphérique. Le flux d'erreur standard (`cerr`) est souvent redirigé vers un fichier et les entrées (`cin`) et sorties (`cout`) standard peuvent être liées à des fichiers ou à des programmes à l'aide de commandes système.

Rediriger signifie envoyer des sorties (ou des entrées) vers un emplacement différent de celui par défaut. La redirection est une fonctionnalité du système d'exploitation, plus qu'une fonction des bibliothèques `iostream`. C++ se contente de donner accès aux quatre flux standard et c'est à l'utilisateur de les rediriger selon ses besoins.

Les opérateurs de redirection de DOS, de Windows et d'Unix sont < (pour les entrées) et > (pour les sorties). Unix offre des possibilités plus évoluées, mais l'idée générale reste la même : prendre une sortie destinée à l'écran pour l'écrire dans un fichier, ou pour la transmettre en entrée d'un programme par un tube. De la même façon, l'entrée d'un programme peut être lue dans un fichier plutôt qu'à partir du clavier.



L'utilisation des tubes permet d'utiliser la sortie d'un programme comme entrée d'un autre.

Les entrées avec *cin*

L'objet global `cin` est responsable des entrées et il est disponible pour votre programme dès lors que vous avez inclus `iostream`. Dans les exemples précédents, vous avez utilisé *l'opérateur d'extraction* (>>) pour placer des données dans les variables du programme. La syntaxe était la suivante :

```
int uneVariable;
cout << "entrez un nombre : ";
cin >> uneVariable;
```

L'objet global `cout` sera présenté plus loin ; pour l'instant, intéressons-nous à la troisième ligne, `cin >> uneVariable;`. Que peut-on en déduire de `cin` ?

`cin` doit forcément être un objet global puisque vous ne l'avez pas défini dans votre code. Vous savez que `cin` surcharge l'opérateur d'extraction (>>) afin qu'il écrive les données de son tampon dans la variable locale `uneVariable`.

Ce qui est moins évident est que `cin` a surchargé cet opérateur pour un grand nombre de paramètres : `int&`, `short&`, `long&`, `double&`, `float&`, `char&`, `char*`, etc. Lorsque vous écrivez `cin >> uneVariable;`, le type de `uneVariable` est évalué. Dans l'exemple précédent, `uneVariable` est un entier, ce qui provoque l'appel de la fonction suivante :

```
istream & operator>> (int &)
```

Le paramètre étant passé par référence, l'opérateur d'extraction peut donc agir sur la variable d'origine. Le Listing 17.1 montre comment l'utiliser.

Listing 17.1 : *cin* gère différents types de données

```
 0:  //Listing 17.1 - cin et les chaînes de caractères
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  int main()
 6:  {
 7:      int unInt;
 8:      long unLong;
 9:      double unDouble;
10:     float myFloat;
11:     unsigned int myUnsigned;
12:
13:     cout << "Int : ";
14:     cin >> unInt;
15:     cout << "Long : ";
16:     cin >> unLong;
17:     cout << "Double : ";
18:     cin >> unDouble;
19:     cout << "Float : ";
20:     cin >> unFloat;
21:     cout << "Unsigned : ";
22:     cin >> unUnsigned;
23:
24:     cout << "\n\nInt :\t" << unInt << endl;
25:     cout << "Long :\t" << unLong << endl;
26:     cout << "Double :\t" << unDouble << endl;
27:     cout << "Float :\t" << unFloat << endl;
28:     cout << "Unsigned :\t" << unUnsigned << endl;
29:     return 0;
30:  }
```

Ce programme produit le résultat suivant :

```
int : 2
Long : 70000
Double : 987654321
Float : 3.33
Unsigned : 25

Int :         2
Long :        70000
Double :      9.87654e+008
Float :       3.33
Unsigned :    25
```

Les lignes 7 à 11 contiennent les déclarations de variables de divers types. Les lignes 13 à 22 demandent à l'utilisateur de fournir des valeur pour ces variables et le résultat est affiché (à l'aide de `cout`) aux lignes 24 à 28.

Le résultat montre que les données ont été enregistrées avec le bon "type" de variable.

Saisie des chaînes

`cin` sait également gérer des pointeurs de caractères (`char*`) ; vous pouvez donc créer un tampon de caractères et utiliser `cin` pour le remplir. Voici un exemple :

```
char votreNom[50]
cout << "Entrez votre nom :";
cin >> votreNom;
```

Si vous tapez `Arthur`, la variable `VotreNom` recevra les caractères A, r, t, h, u, r, \0. Le dernier caractère est le caractère nul, qui est automatiquement ajouté par `cin` : vous devez donc prévoir une taille de tampon suffisante pour stocker la chaîne "utile", plus ce caractère. Le caractère nul signale la "fin de la chaîne" à l'objet `cin`.

Les problèmes des chaînes

Après cette expérience concluante, vous pourriez être surpris de constater que `cin` n'arrive pas à saisir un nom complet. En effet, `cin` considère les espaces comme des caractères de séparation. Lorsqu'il rencontre un blanc ou un retour à la ligne, il suppose donc que la saisie de son paramètre est terminée et, dans le cas d'une chaîne, il ajoute le caractère nul. Le Listing 17.2 illustre ce problème

Listing 17.2 : Tentative de lecture de plusieurs mot à partir de *cin*

```
 0:  //Listing 17.2 - Les chaînes de caractères et cin
 1:
 2:  #include <iostream>
 3:
 4:  int main()
 5:  {
 6:     char VotreNom[50];
 7:     std::cout << "Entrez votre prénom : ";
 8:     std::cin >> VotreNom;
 9:     std::cout << "Votre prénom : " << VotreNom << std::endl;
10:     std::cout << "Entrez vos prénom et nom : ";
11:     std::cin >> VotreNom;
12:     std::cout << "Votre nom complet : " << VotreNom
12a:              << std::endl;
13:     return 0;
14:  }
```

Ce programme produit le résultat suivant :

```
Entrez votre prénom : Bernard
Votre prénom : Bernard
Entrez vos prénom et nom : Bernard Lhermite
Votre nom complet : Bernard
```

La ligne 6 crée un tableau de caractères `VotreNom` pour pouvoir y stocker les saisies de l'utilisateur. La ligne 7 demande à l'utilisateur de saisir son prénom, qui est stocké pour être ensuite affiché.

La ligne 10 demande la saisie d'un nom complet. `cin` lit ce qui est saisi, insère un caractère nul dès qu'il rencontre l'espace entre le prénom et le nom et met fin à la saisie, ce qui n'est pas ce que l'on attendait.

Pour comprendre la raison de ce comportement, examinez le Listing 17.3.

Listing 17.3 : Entrée multiple

```
 0:  // Listing 17.3 - Les chaînes de caractères et cin
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  int main()
 6:  {
 7:     int unInt;
 8:     long unLong;
 9:     double unDouble;
10:     float unFloat;
11:     unsigned int unUnsigned;
12:     char unMot[50];
13:
14:     cout << "int : ";
15:     cin >> unInt;
16:     cout << "Long : ";
17:     cin >> unLong;
18:     cout << "Double : ";
19:     cin >> unDouble;
20:     cout << "Float : ";
21:     cin >> unFloat;
22:     cout << "Mot : ";
23:     cin >> unMot;
24:     cout << "Unsigned : ";
25:     cin >> unUnsigned;
26:
27:     cout << "\n\nInt :\t" << unInt << endl;
```

```
28:     cout << "Long :\t" << unLong << endl;
29:     cout << "Double :\t" << unDouble << endl;
30:     cout << "Float :\t" << unFloat << endl;
31:     cout << "Mot : \t" << unMot << endl;
32:     cout << "Unsigned :\t" << unUnsigned << endl;
33:
34:     cout << "\n\nInt, Long, Double, Float, Mot, Unsigned : ";
35:     cin >> unInt >> unLong >> unDouble;
36:     cin >> unFloat >> unMot >> unUnsigned;
37:     cout << "\n\nInt :\t" << unInt << endl;
38:     cout << "Long :\t" << unLong << endl;
39:     cout << "Double :\t" << unDouble << endl;
40:     cout << "Float :\t" << unFloat << endl;
41:     cout << "Word : \t" << unMot << endl;
42:     cout << "Unsigned :\t" << unUnsigned << endl;
43:
44:     return 0;
45:  }
```

Ce programme produit le résultat suivant :

```
Int : 2
Long : 30303
Double : 393939397834
Float : 3.33
Mot  : Hello
Unsigned : 85

Int :    2
Long :   30303
Double : 3.93939e+011
Float :  3.33
Mot :    Hello
Unsigned :       85

Int, Long, Double, Float, Mot, Unsigned: 3 304938 393847473 6.66 bye -2

Int :    3
Long :   304938
Double : 3.93847e+008
Float :  6.66
Mot :    bye
Unsigned : 4294967294
```

Ce programme crée plusieurs variables, dont un tableau de caractères. L'utilisateur doit fournir des données qui sont ensuite affichées.

La ligne 34 demande toutes ces données d'un seul coup, puis chaque "mot" de la saisie est attribué à la variable appropriée. C'est pour faciliter ce type d'affectations multiples que `cin` doit considérer chaque mot saisi comme une entrée complète destinée à chaque variable. S'il considérait la saisie entière comme celle d'une seule variable, ce type de saisie combinée ne serait pas possible.

Vous pouvez remarquer, à la ligne 42, que le dernier objet demandé était un entier non signé et que l'utilisateur a saisi `-2`. `cin` s'attendant à recevoir un entier non signé, la représentation binaire de `-2` est évaluée comme un entier non signé et `cout` affiche par conséquent la valeur `4294967294`, car c'est cette valeur non signée qui correspond au motif binaire de la valeur signée `-2`.

Plus loin, nous verrons comment saisir une chaîne de plusieurs mots dans un tampon. Pour le moment, nous devons comprendre comment fait l'opérateur surchargé pour gérer cette concaténation des valeurs saisies.

La valeur de retour de *cin*

La valeur renvoyée par l'opérateur d'extraction de `cin` est une référence à un objet `istream`. `cin` étant lui-même un objet `istream`, la valeur renvoyée par cet opérateur peut donc servir pour la lecture suivante :

```
int varUn, varDeux, varTrois;
cout << "Entrez trois nombres :"
cin >> varUn >> varDeux >> varTrois;
```

Dans la dernière ligne, la première lecture est évaluée (`cin >> varUn`) ; on obtient alors un autre objet `istream` auquel on peut appliquer l'opérateur d'extraction pour obtenir la valeur qui sera affectée à `varDeux`. C'est donc comme si vous aviez écrit :

```
((cin >> varUn) >> varDeux) >> varTrois;
```

Vous aurez l'occasion de retrouver cette technique dans le paragraphe consacré à `cout`.

Autres fonctions membres de *cin*

Outre la surcharge de `operator>>()`, `cin` possède un certain nombre d'autres fonctions membres permettant un contrôle plus "fin" des entrées. Ces méthodes permettent d'effectuer les opérations suivantes :

- lire un caractère unique ;
- lire des chaînes ;
- ignorer une entrée ;

- examiner le caractère suivant du tampon ;
- replacer les données dans le tampon.

Saisie d'un caractère unique

Pour lire un caractère à partir de l'entrée standard, vous pouvez utiliser `operator>>()` avec une référence à un caractère. Vous pouvez également utiliser la fonction membre `get()`, qui peut être utilisée de deux façons : sans paramètre (auquel cas, on utilise la valeur renvoyée), ou avec une référence à un caractère.

Utilisation de *get()* sans paramètre

Cette première forme de `get()` renvoie la valeur du caractère obtenu ou `EOF` (*End Of File*, fin de fichier) lorsque la fin du fichier est atteinte. Cette forme est peu utilisée.

À la différence de `operator>>()` qui permet d'obtenir plusieurs valeurs, il n'est pas possible de concaténer cette forme de `get()` pour effectuer des entrées multiples, car la valeur renvoyée n'est pas un objet `iostream`. La ligne de code suivante, par exemple, est donc incorrecte :

```
cin.get() >> maVarUn >> maVarDeux;   // incorrect
```

La valeur renvoyée par (`cin.get() >> maVarUn`) est, en fait, un entier et non un objet `iostream`.

Le Listing 17.4 illustre l'utilisation de `get()` sans paramètres.

Listing 17.4 : Fonction *get()* sans paramètres

```
 0:    // Listing 17.4 -  La fonction get() sans paramètres
 1:
 2:  #include <iostream>
 3:
 4:  int main()
 5:  {
 6:     char ch;
 7:     while ( (ch = std::cin.get()) != EOF)
 8:      {
 9:       std::cout << "ch : " << ch << std::endl;
10:     }
11:     std::cout << "\nTerminé !\n";
12:     return 0;
13:  }
```



Pour sortir de ce programme, vous devrez envoyer un caractère fin de fichier à partir du clavier. Pour cela, faites Ctrl+Z avec DOS et Ctrl+D avec Unix.

Ce programme produit le résultat suivant :

```
Bonjour les amis
ch : B
ch : o
ch : n
ch : j
ch : o
ch : u
ch : r
ch :
ch : l
ch : e
ch : s
ch :
ch : a
ch : m
ch : i
ch : s

^Z (ctrl-z)
Terminé !
```

La ligne 6 déclare une variable locale, `ch`, de type caractère. La boucle `while` affecte les entrées obtenues de `cin.get()` à `ch`, et, s'il ne s'agit pas de EOF, la chaîne est affichée. Cette sortie est, en fait, mise dans un tampon jusqu'à la réception de EOF qui entraîne également la sortie de la boucle.

Notez que cette version de `get()` n'est pas prise en charge dans toutes les implémentations de `istream`, bien qu'elle fasse désormais partie de la norme ANSI/ISO.

Utilisation de *get()* avec une référence caractère

Lorsqu'une variable de type caractère est fournie en paramètre à `get()`, elle reçoit le caractère suivant du flux d'entrée. La valeur renvoyée étant un objet `iostream`, cette forme de `get()` peut donc être enchaînée comme le montre le Listing 17.5.

Listing 17.5 : Utilisation de *get()* avec des paramètres

```
 0:  // Listing 17.5 - Utilisation de get() avec des paramètres
 1:
 2:  #include <iostream>
 3:
 4:  int main()
 5:  {
 6:     char a, b, c;
 7:
 8:     std::cout << "Entrez trois lettres : ";
 9:
10:     std::cin.get(a).get(b).get(c);
11:
12:     std::cout << "a : " << a << "\nb : ";
13:     std::cout << b << "\nc : " << c << std::endl;
14:     return 0;
15:  }
```

Ce programme produit le résultat suivant :

```
Entrez trois lettres : une
a : u
b : n
c : e
```

La ligne 6 crée trois variables caractères, `a`, `b` et `c`, La ligne 10 enchaîne trois appels à `cin.get()`. Le premier, `cin.get(a)` place la première lettre dans `a` et renvoie `cin`, de sorte que l'appel suivant, `cin.get(b)`, place la deuxième lettre dans `b`. Enfin, `cin.get(c)` est appelé et la troisième lettre est placée dans `c`.

Puisque le résultat de `cin.get(a)` est `cin`, vous auriez pu écrire :

```
cin.get(a) >> b;
```

Sous cette forme, `cin.get(a)` renvoie `cin` et le deuxième appel serait donc `cin>>b;`.

Faire	*Ne pas faire*
● Utiliser l'opérateur d'extraction >> lorsque vous voulez arrêter une saisie au premier blanc. ● Utiliser `get()` avec un paramètre de type caractère si vous devez examiner chaque caractère, y compris un blanc.	● Enchaîner les instructions `cin` pour obtenir plusieurs entrées si votre action n'est pas claire. Il vaut mieux utiliser plusieurs commandes plus faciles à comprendre.

Lecture de chaînes à partir de l'entrée standard

L'opérateur d'extraction (>>) peut servir à remplir un tableau de caractère, tout comme la troisième version de la méthode `get()` et de la méthode `getline()`.

Cette forme de `get()` attend trois paramètres :

```
get( pTabChar, tailleFlux, charFin );
```

Le premier (`pTabChar`) est un pointeur sur un tableau de caractères ; le deuxième (`tailleFlux`) est le nombre maximal de caractères à lire, plus un ; le troisième (`charFin`) représente le caractère de fin.

Si le deuxième paramètre vaut `20`, par exemple, `get()` lira 19 caractères et les fera suivre du caractère nul, puis la chaîne sera stockée dans le premier paramètre. Le troisième paramètre, le caractère de fin, vaut '`\n`' (nouvelle ligne) par défaut. Si un caractère de fin est rencontré avant que le nombre maximal de caractères n'ai été lu, un caractère nul est écrit et le caractère de fin reste dans le tampon.

Le Listing 17.6 illustre l'utilisation de cette forme de `get()`.

Listing 17.6 : Utilisation de *get()* avec un tableau de caractères

```
 0:  // Listing 17.6 - Utilisation de get() avec un tableau de caractères
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  int main()
 6:  {
 7:      char chaineUne[256];
 8:      char chaineDeux[256];
 9:
10:     cout << "Entrez la première chaîne : ";
11:     cin.get(chaineUne, 256);
12:     cout << "chaineUne : " << chaineUne << endl;
13:
14:     cout << "Entrez la seconde chaîne : ";
15:     cin >> chaineDeux;
16:     cout << "chaineDeux : " << chaineDeux << endl;
17:     return 0;
18:  }
```

Ce programme produit le résultat suivant :

```
Entrez la première chaîne : C'est l'heure
chaineUne : C'est l'heure
Entrez la seconde chaîne : pour de bon
chaineDeux : pour
```

Les lignes 7 et 8 créent deux tableaux de caractères. La ligne 10 invite l'utilisateur à saisir une chaîne et `cin.get()` est appelée à la ligne 11. Le premier paramètre est le tampon à remplir, le deuxième est le nombre de caractères (plus un) acceptable par `get()` (cette position supplémentaire est utilisée pour placer le caractère nul `'\0'`). Comme on ne fournit pas le troisième paramètre, il s'agit du caractère "Nouvelle ligne".

L'utilisateur saisit "C'est l'heure" suivi d'un saut de ligne, ce qui place cette phrase suivie d'un caractère nul dans `Chaine1`.

L'utilisateur est ensuite invité à entrer une nouvelle chaîne mais, cette fois-ci, à l'aide de l'opérateur d'extraction. Cet opérateur s'arrêtant dès qu'il rencontre un blanc, seule la chaîne `pour` suivie d'un caractère nul sera stockée dans la seconde chaîne ce qui, évidemment, n'était pas ce que l'on souhaitait.

L'utilisation de `get()` avec les trois paramètres est donc une solution parfaite pour lire des chaînes, mais ce n'est pas la seule solution. Un autre moyen de résoudre le problème consiste à utiliser la méthode `getline()`, comme dans le Listing 17.7.

Listing 17.7 : Utilisation de *getline()*

```
0:  // Listing 17.7 - Utilisation de getline()
1:
2:  #include <iostream>
3:  using namespace std;
4:
5:  int main()
6:  {
7:      char chaineUne[256];
8:      char chaineDeux[256];
9:      char chaineTrois[256];
10:
11:     cout << "Entrez la première chaîne : ";
12:     cin.getline(chaineUne, 56);
13:     cout << "chaineUne : " << chaineUne << endl;
14:
15:     cout << "Entrez la seconde chaîne : ";
16:     cin >> chaineDeux;
17:     cout << "chaineDeux : " << chaineDeux << endl;
18:
19:     cout << "Entrez la troisième chaîne : ";
20:     cin.getline(chaineTrois, 256);
21:     cout << "chaineTrois : " << chaineTrois << endl;
22:     return 0;
23:  }
```

Ce programme produit le résultat suivant :

```
Entrez la premiere chaîne : un deux trois
chaineUne : un deux trois
Entrez la deuxieme chaîne : quatre cinq six
chaineDeux : quatre
Entrez la troisieme chaîne : chaineTrois : cinq six
```

Cet exemple mérite un examen attentif et réserve quelques surprises. Les tableaux de caractères sont déclarés lignes 7 à 9.

La ligne 11 demande une chaîne de caractères à l'utilisateur et cette chaîne est lue par la fonction `getline()`. Comme la fonction `get()`, `getline()` prend en paramètre un tampon et un nombre maximal de caractères. Cependant, ici, le retour à la ligne est lu et supprimé du tampon (alors qu'avec `get()` il reste dans le tampon).

La ligne 15 demande une nouvelle chaîne à l'utilisateur. Cette fois, c'est l'opérateur d'extraction qui est utilisé. Dans le résultat, on voit que l'utilisateur a entré `quatre cinq six`, mais que seul le premier mot, `quatre`, a été placé dans `chaineDeux`. Le programme demande ensuite la troisième chaîne à l'utilisateur et la fonction `getline()` est de nouveau appelée. Les mots `cinq six` étant toujours présents dans le tampon d'entrée, ils sont immédiatement lus jusqu'au retour à la ligne. `getline()` se termine et la chaîne dans `chaineTrois` est affichée à la ligne 20.

L'utilisateur n'a eu aucune occasion de saisir la troisième chaîne car le tampon d'entrée contenait déjà des données qui correspondaient à cette requête.

L'opérateur d'extraction de `cin,` à la ligne 16, n'a pas utilisé tout ce qui se trouvait dans le tampon d'entrée : il a lu les caractères jusqu'au premier espace rencontré, puis a stocké le mot ainsi formé dans le tableau de caractères.

`get()` et `getline()`

La fonction membre `get()` est surchargée. Dans la première version, elle ne reçoit aucun paramètre et renvoie la valeur du caractère reçu. Dans la deuxième version, elle prend une référence caractère et renvoie l'objet `istream` par référence.

Dans la troisième et dernière version, `get()` reçoit un tableau de caractères, le nombre de caractères à lire et un caractère de fin (nouvelle ligne par défaut). Cette version de `get()` lit des caractères et les place dans le tableau jusqu'à ce qu'elle en ait lu un de moins que le nombre maximal, ou qu'elle rencontre le caractère de fin. Dans ce cas, ce caractère est laissé dans le tampon d'entrée et la fonction arrête la lecture.

La fonction membre `getline()` reçoit également trois paramètres : le tampon à remplir, le nombre maximal de caractères à lire plus un et le caractère de fin. Cette méthode fonctionne exactement comme `get()`, sauf qu'elle supprime le caractère de fin du tampon d'entrée.

Utilisation de *cin.ignore()*

La fonction membre `ignore()` permet d'ignorer les caractères restants sur une ligne jusqu'à la rencontre d'une fin de ligne (EOL) ou d'une fin de fichier (EOF). Elle attend deux paramètres : le nombre maximal de caractères à ignorer et le caractère de fin. Si vous écrivez `ignore(80, '\n')`, 80 caractères au maximum seront ignorés jusqu'au caractère nouvelle ligne. Ce caractère nouvelle ligne est alors supprimé du tampon d'entrée et la fonction `ignore()` se termine. Le Listing 17.8 illustre l'utilisation de cette méthode.

Listing 17.8 : Utilisation de *ignore()*

```
 0:  // Listing 17.8 - Utilisation de ignore()
 1:  #include <iostream>
 2:  using namespace std;
 3:
 4:  int main()
 5:  {
 6:      char chaineUne[255];
 7:      char chaineDeux[255];
 8:
 9:     cout << "Entrez la première chaîne :";
10:     cin.get(chaineUne, 255);
11:     cout << "chaineUne : " << chaineUne << endl;
12:
13:     cout << "Entrez la deuxième chaîne : ";
14:     cin.getline(chaineDeux, 255);
15:     cout << "chaineDeux : " << chaineDeux << endl;
16:
17:     cout << "\n\nEncore une fois...\n";
18:
19:     cout << "Entrez la première chaîne : ";
20:     cin.get(chaineUne, 255);
21:     cout << "chaineUne : " << chaineUne<< endl;
22:
23:     cin.ignore(255, '\n');
24:
25:     cout << "Entrez la deuxième chaîne : ";
26:     cin.getline(chaineDeux, 255);
27:     cout << "chaineDeux : " << chaineDeux<< endl;
28:     return 0;
29:  }
```

Ce programme produit le résultat suivant :

```
Entrez la première chaîne : Il était une fois
chaineUne :  Il était une fois
```

```
Entrez la deuxième chaîne : chaineDeux :

Encore une fois...
Entrez la première chaîne : Il était une fois
chaineUne : Il était une fois
Entrez la deuxieme chaîne : une princesse mélancolique
chaineDeux : une princesse melancolique
```

Les lignes 6 et 7 créent deux tableaux de caractères. L'utilisateur entre une première chaîne, `Il était une fois`, suivie de la touche Entrée. À la ligne 10, `get()` lit cette chaîne, et la copie dans `chaineUne` jusqu'au caractère nouvelle ligne qui est laissé dans le tampon d'entrée.

La ligne 13 invite à nouveau l'utilisateur à entrer une chaîne, mais c'est `getline()` qui, à la ligne 14, lit le tampon d'entrée jusqu'au caractère nouvelle ligne. Comme il restait une nouvelle ligne dans le tampon à cause de l'appel à `get()` précédent, la lecture s'interrompt immédiatement, avant même que l'utilisateur ait pu entrer quoi que ce soit.

À la ligne 19, l'utilisateur entre à nouveau la première chaîne mais, cette fois, `ignore()` vide le flux d'entrée à la ligne 23, en "mangeant" le caractère nouvelle ligne. Par conséquent, l'appel à `getline()` de la ligne 26 est exécuté et, le tampon d'entrée étant vide, l'utilisateur peut saisir la ligne suivante de l'histoire.

Étudier et renvoyer des caractères : *peek()* et *putback()*

L'objet `cin` dispose de deux méthodes supplémentaires qui sont très pratiques : `peek()`, qui examine mais n'extrait pas le caractère suivant et `putback()`, qui remet un caractère dans le flux d'entrée. Le Listing 17.9 illustre l'utilisation de ces deux fonctions.

Listing 17.9 : Utilisation de *peek()* et de *putback()*

```
 0:    // Illustration des fonctions peek() et putback()
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    char ch;
 7:    cout << "Entrez une phrase : ";
 8:    while ( cin.get(ch) != 0 )
 9:    {
10:       if (ch == '!')
11:          cin.putback('$');
```

```
12:         else
13:            cout << ch;
14:         while (cin.peek() == '#')
15:            cin.ignore(1, '#');
16:     }
17:     return 0;
18:  }
```

Ce programme produit le résultat suivant :

```
Entrez une phrase : Maintenant!c'est#le!moment#de!s'amuser#!
Maintenant$c'estle$momentde$s'amuser$
```

La ligne 6 déclare la variable caractère `ch` et la ligne 7 demande une phrase à l'utilisateur. Le but de ce programme est de transformer les points d'exclamation en dollars et de supprimer les symboles dièse (`#`).

Ce programme boucle des lignes 8 à 16 tant qu'il reçoit des caractères différents d'une fin de fichier (souvenez-vous que `cin.get()` renvoie `0` pour la fin de fichier). Si le caractère courant est un point d'exclamation, il est supprimé et le symbole `$` est remis dans le tampon d'entrée : ce symbole est donc relu par le prochain appel à `cin.get()` dans la boucle. Si le caractère courant n'est pas un point d'exclamation, il est affiché à la ligne 13.

La ligne 14 "examine" le caractère suivant du tampon et, s'il s'agit d'un symbole `#`, il est supprimé à l'aide de la méthode `ignore()` à la ligne 15.

Ce n'est pas l'approche la plus efficace pour réaliser ces deux opérations (en outre, elle ne trouvera pas le symbole dièse s'il est le premier caractère), mais elle illustre bien le fonctionnement de ces deux méthodes.



Les fonctions `peek()` et `putback()` sont généralement utilisées pour l'analyse lexicale des chaînes et des autres données lors de l'écriture d'un compilateur, par exemple.

Les sorties avec *cout*

Vous avez déjà utilisé `cout` pour écrire des chaînes, des entiers et d'autres données numériques sur l'écran avec l'opérateur `<<`. Il est également possible de formater les données, d'aligner des colonnes et d'écrire les données numériques en décimal ou en hexadécimal.

Purger la sortie

Vous avez vu que `endl` écrit une nouvelle ligne et permet de purger le tampon de sortie. `endl` appelle la méthode `flush()` de `cout`, qui écrit toutes les données contenues dans le tampon de sortie. Vous pouvez également appeler directement la méthode `flush()`, soit en l'invoquant sur l'objet `cout`, soit à l'aide de l'instruction suivante :

```
cout << flush();
```

Cette méthode est pratique lorsque vous voulez vous assurer que le tampon a été vidé et que son contenu a été écrit à l'écran.

Fonctions de sortie

Tout comme l'opérateur d'extraction (`>>`) est complété par les méthodes `get()` et `getline()`, *l'opérateur d'insertion* (`<<`) peut être complété par les méthodes `put()` et `write()`.

Écrire des caractères avec *put()*

`put()` permet d'écrire un caractère unique vers le flux de sortie. Cette fonction renvoie une référence à un `ostream` et, `cout` étant lui-même un objet `ostream`, vous pouvez donc enchaîner les appels à `put()` comme vous pouvez le faire avec l'opérateur d'insertion. L'instruction suivante, par exemple, permet d'afficher la chaîne `Hello` à l'écran :

```
cout.put('H').put('e').put('l').put('l').put('o').put('\n');
```



Certains compilateurs non conformes à la norme auront des problèmes avec ce code. Si votre compilateur n'affiche pas le mot `Hello`, utilisez une autre méthode.

Écrire des caractères avec *write()*

La fonction `write()` fonctionne exactement comme l'opérateur `<<`, sauf qu'elle accepte un paramètre indiquant le nombre maximal de caractères à écrire :

```
cout.write(Texte, Taille)
```

Le premier paramètre est le texte à afficher. Le deuxième correspond au nombre de caractères de `Texte` qui seront affichés. Ce nombre peut être inférieur ou supérieur à la taille réelle de `Texte` : s'il est supérieur, vous afficherez également les valeurs qui se trouvent en mémoire à la suite de `Texte`. Le Listing 17.10 illustre l'utilisation de cette méthode.

Listing 17.10 : Utilisation de *write()*

```
 0:  // Listing 17.10 - Utilisation de write()
 1:  #include <iostream>
 2:  #include <string.h>
 3:  using namespace std;
 4:
 5:  int main()
 6:  {
 7:     char Une[] = "Qui vivra verra";
 8:
 9:     int complet = strlen(Une);
10:     int tropCourt = complet - 4;
11:     int tropLong = complet + 6;
12:
13:     cout.write(Une, complet) << endl;
14:     cout.write(Une, tropCourt) << endl;
15:     cout.write(Une, tropLong) << endl;
16:     return 0;
17:  }
```

Ce programme produit le résultat suivant :

```
Qui vivra verra
Qui vivra v
Qui vivra verra !?mxI
```



La dernière ligne sera probablement différente, car elle accède à une zone mémoire qui ne fait pas partie de la variable initialisée.

Ce listing affiche une phrase mais, à chaque fois, elle en affiche une quantité différente. La phrase est créée à la ligne 7. À la ligne 9, l'entier `complet` reçoit la longueur de la phrase à l'aide de fonction globale `strlen()` incluse grâce à la directive de la ligne 2. Deux autres longueur seront également utilisées : `tropCourt` est définie avec cette longueur moins quatre, alors que `tropLong` est définie avec cette longueur plus six.

La phrase complète est affichée à la ligne 13 à l'aide de la fonction `write()`. La longueur correspondant à la longueur réelle de la phrase, celle-ci s'affiche donc correctement.

La phrase est à nouveau affichée à la ligne 14, mais avec quatre caractères de moins.

La ligne 15 affiche de nouveau la phrase, mais on demande cette fois-ci à la fonction `write()` d'écrire six caractères de plus que la longueur de cette phrase. Les six octets qui sont stockés en mémoire à la suite de la phrase sont donc écrits. Cette mémoire pouvant contenir n'importe quoi, votre résultat sera donc sûrement différent de ce qui est présenté.

Manipulateurs, indicateurs d'état et instructions de formatage

Le flux de sortie possède un certain nombre d'indicateurs d'état, qui déterminent la base à utiliser (décimale ou hexadécimale), la largeur des champs et le caractère de remplissage des champs. Un indicateur d'état est un simple octet dont chaque bit a une signification particulière (le Chapitre 21 traitera de la manipulation des bits). Chaque indicateur de `ostream` peut être initialisé à l'aide de fonctions membres et de manipulateurs.

Utilisation de *cout.width()*

La largeur par défaut de la sortie sera toujours ajustée pour permettre d'afficher les nombres, les caractères ou les chaînes du tampon de sortie. Vous pouvez la modifier avec la fonction `width()`, qui doit être appelée sur un objet `cout` car il s'agit d'une fonction membre. Elle n'agit que sur le champ de sortie suivant ; la largeur reprend ensuite immédiatement sa valeur par défaut.

Listing 17.11 : Modification de la largeur de la sortie

```
 0:  // Listing 17.11 - Modification de la taille de la sortie
 1:  #include <iostream>
 2:  using namespace std;
 3:
 4:  int main()
 5:  {
 6:      cout << "Début >";
 7:      cout.width(25);
 8:      cout << 123 << "< Fin\n";
 9:
10:     cout << "Début >";
11:     cout.width(25);
12:     cout << 123 << "< Suivant >";
13:     cout << 456 << "< Fin\n";
14:
15:     cout << "Début >";
16:     cout.width(4);
17:     cout << 123456 << "< Fin\n";
18:
19:     return 0;
20:  }
```

Ce programme produit le résultat suivant :

```
Début >                    123< Fin
Début >                    123< Suivant >456< Fin
Début >123456< Fin
```

Les lignes 6 à 8 affichent `123` sur la première ligne, dans un champ dont la largeur a été définie à 25 caractères (ligne 7).

La deuxième ligne affiche la valeur `123` dans le même champ que la ligne précédente, puis la valeur `456`. Vous remarquerez que cette dernière valeur est affichée dans un champ dont la largeur a été ajustée à celle de la valeur ; comme on l'a indiqué précédemment, l'effet de `width()` n'agit que sur la sortie suivante, pas sur les autres.

La dernière ligne vous montre que le fait de définir un champ plus petit que la sortie donne le même résultat que s'il était défini à sa taille exacte. Une largeur trop petite ne tronquera donc pas ce qui est affiché.

Définition des caractères de remplissage

Les caractères de remplissage par défaut de `cout` sont des blancs, comme vous avez pu le constater dans l'exemple précédent. Si vous voulez remplacer ces blancs par un autre caractère, comme des astérisques, vous devez appeler `fill()` en lui passant le caractère que vous voulez utiliser. En modifiant les instructions de l'exemple précédent de cette façon :

```
cout << "Début >";
cout.width(25);
cout.fill('*');
cout << 123 << "< Fin\n";
```

On obtient l'affichage de la ligne suivante :

```
Début >**********************123< Fin
```

Gérer l'état des résultats : définition des indicateurs

On dit que les objets ont un *état* quand une partie de leurs données, ou la totalité, représente une *condition* qui peut changer pendant l'exécution du programme. Vous pouvez indiquer, par exemple, s'il faut afficher les zéros de droite (pour que 20,00 ne soit pas tronqué en 20).

Les objets `iostream` mémorisent leur état grâce à des indicateurs. Vous pouvez les définir en appelant la fonction `setf()` et en lui passant l'une ou l'autre des constantes énumérées prédéfinies. L'appel `setf(ios::showpoint)`, par exemple, demande l'affichage des zéros de droite.

Les constantes énumérées font partie de la classe `iostream` (`ios`). Il faut donc, lorsqu'on les utilise avec `setf()`, les appeler avec leur qualification complète `ios::`*`nomIndicateur`*, comme dans `ios::showpoint`. Le Tableau 17.1 montre quelques indicateurs.

Pour les utiliser, vous devez inclure `iostream` dans votre programme. En outre, pour les indicateurs qui prennent des paramètres, vous devez également inclure `iomanip`.

Tableau 17.1 : Certains indicateurs *iostream*

Indicateur	*Objectif*
`showpoint`	Affiche un point décimal et les zéros de droite exigés par la précision.
`showpos`	Active le signe plus (+) devant les nombres positifs.
`left`	Aligne le résultat à gauche.
`right`	Aligne le résultat à droite.
`internal`	Aligne le signe d'un nombre à gauche et sa valeur à droite.
`scientific`	Affiche les valeurs à virgule flottante en notation scientifique.
`fixed`	Affiche les valeurs à virgule flottante en notation décimale.
`showbase`	Ajoute "`0x`" devant les nombres hexadécimaux.
`Uppercase`	Affiche les nombres hexadécimaux et scientifiques en majuscules.
`dec`	Fixe la base des nombres pour un affichage en décimal.
`oct`	Fixe la base des nombres pour un affichage en octal (base 8).
`hex`	Fixe la base des nombres pour un affichage en hexadécimal (base 16).

Ces indicateurs peuvent également être concaténés avec l'opérateur d'insertion.

Le Listing 17.12 montre un exemple et introduit également le manipulateur `setw`, qui fixe la largeur, mais peut aussi être concaténé avec l'opérateur d'insertion.

Listing 17.12 : Utilisation de *setf*

```
0:  // Listing 17.12 - Utilisation de setf
1:  #include <iostream>
2:  #include <iomanip>
3:  using namespace std;
4:
5:  int main()
6:  {
7:      const int nombre = 185;
8:      cout << "Nombre  " << nombre << endl;
9:
```

```
10:     cout << "Nombre  " << hex << nombre << endl;
11:
12:     cout.setf(ios::showbase);
13:     cout << "Nombre  " << hex << nombre << endl;
14:
15:     cout << "Nombre  " ;
16:     cout.width(10);
17:     cout << hex << nombre << endl;
18:
19:     cout << "Nombre  " ;
20:     cout.width(10);
21:     cout.setf(ios::left);
22:     cout << hex << nombre << endl;
23:
24:     cout << "Nombre  " ;
25:     cout.width(10);
26:     cout.setf(ios::internal);
27:     cout << hex << nombre << endl;
28:
29:     cout << "Nombre  " << setw(10) << hex << nombre << endl;
30:     return 0;
31: }
```

Ce programme donne le résultat suivant :

```
Nombre  185
Nombre  b9
Nombre  0xb9
Nombre          0xb9
Nombre  0xb9
Nombre  0x         b9
Nombre  0x         b9
```

La ligne 7 initialise la constante entière `nombre` avec la valeur `185` et la ligne suivante l'affiche normalement.

Cette valeur est à nouveau affichée à la ligne 10, mais le manipulateur `hex` est concaténé, ce qui provoque l'affichage en hexadécimal, soit `b9`.

La ligne 12 définit l'indicateur `showbase`. Le préfixe `0x` sera donc ajouté à tous les nombres hexadécimaux comme le montre l'affichage.

La ligne 16 fixe la largeur à 10 et, par défaut la valeur est alignée à droite. À la ligne 20, la largeur est de nouveau 10, mais l'alignement est défini à gauche (*left*).

À la ligne 25, la largeur est toujours 10, mais avec un alignement `internal`. `0x` est donc aligné à gauche, mais `b9` est aligné à droite.

Enfin, à la ligne 29, le manipulateur `setw()` est utilisé pour définir la largeur à 10 et la valeur est à nouveau affichée.

Dans ce listing, vous remarquerez que si les indicateurs sont utilisés dans la liste `cout`, ils n'ont pas besoin d'être qualifiés : `hex` peut être passé tel quel. Par contre, lorsque vous utilisez la fonction `setf()`, vous devez qualifier les indicateurs avec la classe ; `hex` est passé sous la forme `ios::hex`. Vous pouvez constater cette différence entre les lignes 17 et 21.

Flux ou fonction *printf()*

La plupart des implémentations de C++ fournissent également les bibliothèques standard des E/S de C, qui contiennent notamment la fonction `printf()`. Celle-ci est parfois plus facile à utiliser que `cout`, mais son emploi est déconseillé.

`printf()` ne vérifiant pas les types, vous pouvez donc lui demander d'afficher un entier comme s'il s'agissait d'un caractère, et *vice versa*. Elle ne gère pas non plus les classes et il n'est donc pas possible de lui apprendre comment afficher les données de vos classes : vous devrez fournir chaque membre de la classe à `printf()`, un par un.

Nous allons cependant passer rapidement en revue l'utilisation de cette fonction car beaucoup d'anciens programmes l'utilisent. Cependant, comme nous l'avons déjà dit, il est déconseillé de l'utiliser dans vos programmes C++.

Pour utiliser `printf()`, il faut inclure le fichier en-tête `stdio.h`. Sous sa forme la plus simple, cette fonction reçoit une chaîne de format comme premier paramètre, puis une série de valeurs pour les autres paramètres.

La chaîne de format est une chaîne pouvant contenir du texte et des spécificateurs de conversion, le tout entre guillemets. Les spécificateurs de conversion, dont les plus utilisés sont présentés dans le Tableau 17.2, commencent tous par le symbole pour cent (`%`).

Tableau 17.2 : Spécificateurs de conversions courants

Spécificateurs	*Utilisé pour*
`%s`	Chaînes
`%d`	Entiers
`%l`	Entier `long`
`%ld`	`double`
`%f`	`float`

Chaque spécificateur de conversion peut également fournir des indications de taille et de précision, exprimées sous la forme d'un nombre à virgule flottante. Les chiffres situés à gauche du point décimal représentent la largeur totale, ceux de droite la précision pour les nombres à virgule flottante. Ainsi, `%5d` est le spécificateur pour un entier de cinq chiffres et `%15.5f` celui d'un nombre à virgule flottante de 15 chiffres, dont 5 pour la partie décimale. Le Listing 17.13 illustre diverses utilisations de la fonction `printf()`.

Listing 17.13 : Utilisation de *printf()*

```
 0:  //17.13 Utilisation de printf()
 1:  #include <stdio.h>
 2:
 3:  int main()
 4:  {
 5:      printf("%s", "Bonjour\n");
 6:
 7:      char *phrase = "Bonjour tout le monde !\n";
 8:      printf("%s", phrase);
 9:
10:     int x = 5;
11:     printf("%d\n", x);
12:
13:     char *phraseDeux = "Voici des valeurs :   ";
14:     char *phraseTrois = " et quelques autres : ";
15:     int y = 7, z = 35;
16:     long longVar = 98456;
17:     float floatVar =  8.8f;
18:
19:     printf("%s %d %d", phraseDeux, y, z);
20:     printf("%s %ld %f\n",phraseTrois, longVar, floatVar);
21:
22:     char *phraseQuatre = "Version mise en forme : ";
23:     printf("%s %5d %10d  %10.5f\n",
23a:            phraseQuatre, y, z, floatVar);
24:
25:     return 0;
26: }
```

Ce programme produit le résultat suivant :

```
Bonjour
Bonjour tout le monde !
5
Voici des valeurs :    7 35   et quelques autres : 98456 8.800000
Version mise en forme :      7           35     8.800000
```

La première instruction `printf()`, à la ligne 5, utilise la forme standard : elle prend en paramètre une chaîne entre guillemets contenant un spécificateur de conversion et une valeur à insérer à la place de ce spécificateur.

Le `%s` indique qu'il s'agit d'une chaîne et, ici, cette chaîne est un littéral entre guillemets : "Bonjour".

La deuxième instruction (ligne 8) est comme la première, mais plutôt que d'indiquer la chaîne de caractères entre guillemets, on utilise un pointeur de caractère. Le résultat est toutefois identique.

La troisième instruction `printf()`, ligne 11, utilise le spécificateur de conversion en entier (`%d`) et la variable entière `x`. La quatrième instruction, ligne 19, est plus complexe car elle concatène trois valeurs. La chaîne de format contenant les trois spécificateurs de conversion est suivie des trois valeurs, séparées par des virgules. La ligne 20 est identique à la ligne 19, mais les valeurs utilisées sont différentes.

L'instruction de la ligne 23, enfin, utilise des spécificateurs de format pour indiquer la largeur et la précision.

Il ne faut pas oublier que cette instruction ne contrôle pas le type des données et que `printf()` ne peut être déclarée comme une fonction amie ou comme une fonction membre d'une classe.

Manipulation des sorties : résumé

Pour formater les sorties en C++, on utilise une combinaison de caractères spéciaux, de manipulateurs et d'indicateurs.

Les caractères spéciaux suivants sont inclus dans une chaîne en sortie envoyée à `cout` à l'aide de l'opérateur d'insertion :

```
\n. Nouvelle ligne.
\r. Retour chariot.
\t. Tabulation.
\\. Antislash.
\ddd (nombre octal). Caractère ASCII.
\a. Alarme (sonore).
```

Par exemple :

```
cout << "\aUne erreur s'est produite\t"
```

émet un bip, affiche un message d'erreur et se positionne sur la prochaine tabulation. Les manipulateurs sont utilisés avec `cout`. Ceux qui attendent des paramètres nécessitent l'inclusion de `iomanip` dans votre fichier.

Les opérateurs suivants ne requièrent *pas* `iomanip` :

- `flush`. Purge le tampon de sortie.
- `endl`. Insère une nouvelle ligne et purge le tampon de sortie.
- `oct`. Fixe la base octale en sortie.
- `dec`. Fixe la base décimale en sortie.
- `hex`. Fixe la base hexadécimale en sortie.

Les opérateurs suivants *requièrent* `iomanip` :

- `setbase (base)`. Fixe la base en sortie (0 = décimal, 8 = octal, 10 = décimal, 16 = hexa).
- `setw (largeur)`. Fixe la largeur minimale du champ en sortie.
- `setfill (car)`. Caractère de remplissage à utiliser lorsque *largeur* est défini.
- `setprecision (p)`. Fixe la précision pour les nombres à virgule flottante.
- `setiosflags (f)`. Définit un ou plusieurs indicateurs `ios`.
- `resetiosflags (f)`. Redéfinit un ou plusieurs indicateurs `ios`.

Par exemple :

```
cout << setw(12) << setfill('#') << hex << x << endl;
```

fixe à 12 la largeur du champ et le `'#'` comme caractère de remplissage, précise une sortie hexa, affiche la valeur de `x`, place une nouvelle ligne dans le tampon et le purge. Tous les manipulateurs, sauf `flush`, `endl` et `setw`, restent en place jusqu'à ce qu'ils soient modifiés ou jusqu'à la fin du programme. `setw` revient à sa valeur par défaut après le traitement `cout` en cours.

Un certain nombre d'indicateurs peuvent être utilisés avec les manipulateurs `setiosflags` et `resetiosflags`. Ils ont été énumérés au Tableau 17.1.

Des informations supplémentaires peuvent être obtenues dans le fichier `ios` et la documentation de votre compilateur.

Entrées/sorties sur fichier

Les flux offrent un moyen uniforme de traitement des données provenant du clavier ou du disque dur, ou allant vers l'écran ou le disque dur. Dans tous les cas, vous pouvez utiliser les opérateurs d'insertion, d'extraction ou les autres fonctions et manipulateurs connexes. Pour ouvrir et fermer des fichiers, vous devez créer des objets `ifstream` et `ofstream`.

ofstream et *ifstream*

Les objets utilisés pour lire ou écrire dans des fichiers sont des objets `ifstream` et `ofstream`. Ces classes dérive de la classe `iostream` que vous avez déjà utilisée.

Avant d'écrire dans un fichier, vous devez d'abord créer un objet `ofstream`, puis lui associer un fichier sur disque. Pour utiliser des objets `ofstream`, votre programme doit inclure `fstream`.



Vous n'avez pas besoin d'inclure explicitement le fichier `iostream` puisqu'il est compris dans le fichier `fstream`.

État des E/S

Les objets `iostream` gèrent des indicateurs qui précisent l'état de votre entrée et de votre sortie. Vous pouvez contrôler ces indicateurs à l'aide des fonctions booléennes `eof()`, `bad()`, `fail()` et `good()`. La fonction `eof()` renvoie la valeur `true` si l'objet `iostream` a rencontré `EOF` (Fin de fichier). La fonction `bad()` renvoie `true` si vous tentez une opération non valide. La fonction `fail()` renvoie `true` dès que `bad()` renvoie `true` ou qu'une opération échoue. La fonction `good()`, enfin, renvoie `true` lorsque les trois autres opérations renvoient `false`.

Ouverture des fichiers pour les E/S

Pour utiliser un fichier, vous devez d'abord l'ouvrir. Pour ouvrir le fichier `monFichier.cpp` avec un objet `ofstream`, il faut déclarer une instance d'un objet `ofstream` et lui passer le nom de fichier en paramètre :

```
ofstream fout("monFichier.cpp");
```

Cette instruction tente d'ouvrir le fichier `monFichier.cpp` en sortie. L'ouverture de ce fichier en entrée fonctionne de la même façon, mais avec un objet `ifstream` :

```
ifstream fin("monFichier.cpp");
```

`fout` et `fin` ont été choisis ici en raison de leur similitude avec `cout` et `cin`. Les noms pourraient aussi refléter le contenu du fichier accédé.

Chaque objet flux que vous créez ouvre un fichier en lecture (en entrée), en écriture (en sortie) ou les deux. Il est important de refermer ce fichier en fin de traitement à l'aide de la fonction `close()` ; cette opération garantit que le fichier ne sera pas corrompu et que les données que vous avez écrites sont bien transférées sur le disque.

Lorsque les objets flux ont été associés à des fichiers, ils peuvent être utilisés comme n'importe quel autre objet flux. Le Listing 17.14 illustre ces manipulations.

Listing 17.14 : Ouverture de fichiers en lecture et en écriture

```
 0:  //Listing 17.14 Ouverture de fichiers en lecture et écriture
 1:  #include <fstream>
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  int main()
 6:  {
 7:     char nomFichier[80];
 8:     char tampon[255];    // pour l'entrée utilisateur
 9:     cout << "Nom du fichier : ";
10:     cin >> nomFichier;
11:
12:     ofstream fout(nomFichier);  // ouvrir fout en écriture
13:
13a:    << "Cette ligne est écrite directement dans le
13b:       fichier...\n";
14:     cout << "Entrez le texte destiné au fichier : ";
15:     // supprimer la nouvelle ligne après le nom de fichier
15a:    cin.ignore(1, '\n');
16:     // récupérer la saisie de l'utilisateur
16a:    cin.getline(tampon, 255);
17:     fout << tampon << "\n";   // et l'écrire dans le fichier
18:     // fermer le fichier, prêt pour réouverture
18a:    fout.close();
19:
20:     ifstream fin(nomFichier);    // rouvrir en lecture
21:     cout << "Contenu du fichier :\n";
22:     char ch;
23:     while (fin.get(ch))
24:        cout << ch;
25:
26:     cout << "\n***Fin du fichier.***\n";
27:
28:     fin.close();      // il faut toujours fermer le fichier!
29:     return 0;
30:  }
```

Ce programme produit le résultat suivant :

```
Nom du fichier : test1
Entrez le texte destiné au fichier : Voila ce que je voulais dire
Contenu du fichier :
Cette ligne est écrite directement dans le fichier...
Voila ce que je voulais dire

***Fin du fichier.***
```

La ligne 7 crée un tableau de caractères pour stocker le nom du fichier et la ligne 8 crée un autre tampon pour recevoir la saisie de l'utilisateur. Ce dernier est invité à entrer un nom de fichier sur la ligne suivante, qui est stocké dans le tampon `nomFichier`. La ligne 12 crée un objet `ofstream`, `fout`, et l'associe au nom du fichier. Cette ligne ouvre le fichier en écriture ; s'il existait déjà, son contenu est supprimé.

La ligne 13 écrit directement dans le fichier une chaîne de texte La ligne suivante demande à l'utilisateur de saisir une chaîne. Le caractère nouvelle ligne de l'entrée utilisateur précédente (nom du fichier) qui restait dans le tampon d'entrée est supprimé à la ligne 15 par la fonction `ignore()`, que nous avons vue précédemment. La nouvelle saisie de l'utilisateur est ensuite stockée dans `tampon`. La ligne 17 écrit son contenu puis le caractère nouvelle ligne dans le fichier. La ligne suivante ferme le fichier.

La ligne 20 réouvre le fichier, cette fois en entrée avec `ifstream`, et le contenu est lu caractère par caractère aux lignes 23 et 24.

Modification du comportement par défaut de *ofstream* à l'ouverture

Par défaut, lorsque vous ouvrez un fichier en écriture, celui-ci est automatiquement créé s'il n'existe pas. Dans le cas contraire, son contenu est supprimé. Si ce comportement par défaut ne vous convient pas, vous pouvez transmettre explicitement un second paramètre au constructeur de votre objet `ofstream`. Voici les valeurs que vous pouvez utiliser :

- `ios::app`. Ajoute les données à la fin du fichier existant au lieu de supprimer son contenu.
- `ios::ate`. Vous place en fin de fichier, mais vous pouvez écrire n'importe où dans ce fichier.
- `ios::trunc`. Comportement par défaut. Le fichier existant est vidé.
- `ios::nocreate`. Échec de l'ouverture si le fichier n'existe pas.
- `ios::noreplace`. Échec de l'ouverture si le fichier existe déjà.
- Le Listing 17.15 illustre l'ajout de données à un fichier existant en réouvrant celui du listing précédent pour lui ajouter du texte.

Listing 17.15 : Ajout de données en fin de fichier

```
0:  //Listing 17.15 Ajout de données en fin de fichier
1:  #include <fstream>
2:  #include <iostream>
3:  using namespace std;
4:
```

```
 5:  int main()   // renvoie 1 si erreur
 6:  {
 7:     char nomFichier[80];
 8:     char tampon[255];
 9:     cout << "Entrez de nouveau le nom du fichier : ";
10:     cin >> nomFichier;
11:
12:     ifstream fin(nomFichier);
13:     if (fin)                // existe déjà ?
14:     {
15:        cout << "Contenu actuel du fichier :\n";
16:        char ch;
17:        while (fin.get(ch))
18:           cout << ch;
19:        cout << "\n***Fin du fichier.***\n";
20:     }
21:     fin.close();
22:
23:     cout << "\nOuverture de " << nomFichier
23a:          << " en mode ajout...\n";
24:
25:     ofstream fout(nomFichier, ios::app);
26:     if (!fout)
27:     {
28:        cout << " Ouverture impossible de "
28a:             << nomFichier << " en ajout.\n";
29:        return(1);
30:     }
31:
32:     cout << "\nEntrez le texte à ajouter au fichier : ";
33:     cin.ignore(1, '\n');
34:     cin.getline(tampon, 255);
35:     fout << tampon << "\n";
36:     fout.close();
37:
38:     fin.open(nomFichier); // réaffecte l'objet fin existant !
39:     if (!fin)
40:     {
41:        cout << "Ouverture impossible de "
41a:             << nomFichier << " en lecture.\n";
42:        return(1);
43:     }
44:     cout << "\nContenu du fichier :\n";
45:     char ch;
```

```
46:     while (fin.get(ch))
47:        cout << ch;
48:     cout << "\n***Fin du fichier.***\n";
49:     fin.close();
50:     return 0;
51:  }
```

Ce programme produit le résultat suivant :

```
Entrez de nouveau le nom du fichier : test1
Contenu actuel du fichier :
Cette ligne est écrite directement dans le fichier...
Voila ce que je voulais dire

***Fin du fichier.***

Ouverture de test1 en mode ajout...
Entrez le texte à ajouter au fichier : mais hélas...

Contenu du fichier :
Cette ligne est écrite directement dans le fichier
Voila ce que je voulais dire
mais hélas...

***Fin du fichier.***
```

Comme dans le précédent listing, ce programme demande un nom de fichier à l'utilisateur aux lignes 9 et 10. Cette fois-ci, la ligne 12 crée un objet flux de fichier en entrée. L'ouverture du fichier est testée ligne 13 et, si le fichier existe, son contenu est affiché aux lignes 15 à 19. Notez que `if (fin)` est équivalent à `if (fin.good())`.

Le fichier est ensuite fermé puis rouvert à la ligne 25, cette fois-ci en mode ajout. Le fichier est testé après chaque ouverture pour s'assurer que cette opération s'est déroulée correctement. Ici, `if (!fout)` est équivalent à `if (fout.fail())`. Si le fichier n'a pas pu être ouvert, le programme affiche un message d'erreur en ligne 28 et se termine par l'instruction `return`. Si l'ouverture a réussi, l'utilisateur est invité à saisir le texte à ajouter, puis le fichier est fermé à la ligne 36.

Enfin, comme dans le listing précédent, on réouvre le fichier en mode lecture. Cette fois-ci, cependant, il n'y a pas besoin de redéclarer `fin` : on lui attribue simplement le même nom de fichier. L'ouverture du fichier est testée ligne 39. Si tout s'est bien passé, le contenu du fichier est affiché et le fichier est fermé four la dernière fois.

Faire	*Ne pas faire*
● Tester le bon déroulement de chaque ouverture de fichier. ● Réutiliser les objets `ifstream` et `ofstream` existants. ● Fermer tous les objets `fstream` quand vous n'en avez plus besoin.	● Essayer de fermer ou de réaffecter `cin` ou `cout`. ● Utiliser `printf()` dans vos programmes C++ si ce n'est pas essentiel.

Fichiers binaires ou fichiers texte

Certains systèmes d'exploitation différencient les fichiers texte des fichiers binaires. Les fichiers texte permettent de stocker toutes les données sous la forme texte. Des nombres tels que 54,325 seront donc stockés comme une chaîne de chiffres ('`5`', '`4`', ',', '`3`', '`2`', '`5`'). Ce n'est pas très efficace, mais cela présente l'avantage d'être lu par des programmes simples, comme les fichiers de commandes DOS ou Windows.

Pour aider le système de fichiers à distinguer les fichiers texte des fichiers binaires, C++ fournit l'indicateur `ios::binary`. Cet indicateur est ignoré par les nombreux systèmes qui stockent toutes leurs données en format binaire. Il peut même être refusé par certains compilateurs !

Les fichiers binaires stockent non seulement des entiers et des chaînes, mais également des structures de données entières. Vous pouvez écrire toutes les données en une seule fois grâce à la méthode `write()() fstream`.

Les données écrites avec `write()`, peuvent être relues avec `read()`. Ces deux méthodes attendant qu'on leur passe un pointeur de caractère comme premier paramètre, vous devrez donc transtyper l'adresse de la classe en pointeur de caractère si vous souhaitez écrire ou lire un objet.

Le deuxième paramètre de ces méthodes est le nombre de caractères à lire ou à écrire, que vous pouvez déterminer avec l'opérateur `sizeof()`. Notez que, dans le cas des objets, on écrit que les données, pas les méthodes ; de même, on ne relit que les données. Le Listing 17.16 montre comment stocker le contenu d'un objet dans un fichier.

Listing 17.16 : Écriture d'un objet dans un fichier

```
0:  //Listing 17.16 Écriture d'un objet dans un fichier
1:  #include <fstream>
2:  #include <iostream>
3:  using namespace std;
4:
```

```
 5:  class Animal
 6:  {
 7:  public:
 8:      Animal(int poids, long jours):
8a:       sonPoids(poids), saDureeVie(jours) {}
 9:      ~Animal(){}
10:
11:     int GetPoids()const { return sonPoids; }
12:     void SetPoids(int poids) { sonPoids = poids; }
13:
14:     long GetDureeVie()const { return saDureeVie; }
15:     void SetDureeVie (long jours) { saDureeVie = jours; }
16:
17:  private:
18:     int sonPoids;
19:     long saDureeVie;
20:  };
21:
22:  int main()   // renvoie 1 si erreur
23:  {
24:     char nomFichier[80];
25:
26:
27:     cout << "Entrez le nom du fichier : ";
28:     cin >> nomFichier;
29:     ofstream fout(nomFichier, ios::binary);
30:     if (!fout)
31:     {
32:        cout << "Impossible d'ouvrir "
32a:           << nomFichier<< " en écriture.\n";
33:        return(1);
34:     }
35:
36:     Animal Ours(50, 100);
37:     fout.write((char*) &Ours, sizeof Ours);
38:
39:     fout.close();
40:
41:     ifstream fin(nomFichier, ios::binary);
42:     if (!fin)
43:     {
44:        cout << "Impossible d'ouvrir "
44a:           << nomFichier << " en lecture.\n";
45:        return(1);
46:     }
47:
48:     Animal Ours2(1, 1);
```

```
49:
50:     cout << "Poids Ours2 : " << Ours2.GetPoids() << endl;
51:     cout << "Jours Ours2 : " << Ours2.GetDureeVie() << endl;
52:
53:     fin.read((char*) &Ours2, sizeof Ours2);
54:
55:     cout << "Poids Ours2 : " << Ours2.GetPoids() << endl;
56:     cout << "Jours Ours2 : " << Ours2.GetDureeVie() << endl;
57:     fin.close();
58:     return 0;
59:  }
```

Ce programme produit le résultat suivant :

```
Entrez le nom du fichier: Animaux
Poids Ours2 : 1
Jours Ours2 : 1
Poids Ours2 : 50
Jours Ours2 : 100
```

La classe `Animal` est déclarée aux lignes 5 à 20. Les lignes 24 à 34 créent un fichier et l'ouvrent pour une sortie en mode binaire. Un animal d'un poids de 50 kg et d'un âge de 100 jours est créé à la ligne 36, ses données sont écrites dans le fichier à la ligne 37.

Le fichier est fermé à la ligne 39, puis réouvert pour une lecture en mode binaire à la ligne 41. La ligne 48 crée un second animal à la ligne 48, avec un poids de 1 kg et un âge de 1 jour. Les données du fichiers sont lues et placées dans ce nouvel animal à la ligne 53 ; elles écrasent donc les anciennes valeurs, ce que l'on peut vérifier par ce qui est affiché par le programme.

Traitement de la ligne de commande

De nombreux systèmes d'exploitation, comme DOS ou Unix, permettent à l'utilisateur de passer des paramètres au lancement du programme. Ces paramètres sont appelés options de la ligne de commande et sont séparés par des espaces. Voici un exemple :

```
UnProgramme Param1 Param2 Param3
```

Ces paramètres ne sont pas directement transmis à `main()`. En effet, cette fonction reçoit deux paramètres. Le premier est un entier qui représente le nombre de paramètres de la ligne de commande, y compris le nom du programme : chaque programme a donc toujours au moins un paramètre. L'exemple ci-dessus contient quatre paramètres (le nom du programme, plus trois paramètres en ligne de commande).

Le second paramètre est un tableau de pointeurs sur des chaînes de caractères. Le nom d'un tableau étant lui-même un pointeur sur le premier élément de ce tableau, vous pouvez donc déclarer ce paramètre comme étant un pointeur de pointeur de caractères, un pointeur sur un tableau de caractères, ou un tableau de tableaux de caractères.

Le premier argument est souvent appelé `argc` (*argument count)* et le second `argv` (*argument vector)*, mais vous pouvez adopter votre propre convention.

Il est courant de tester `argc` pour être sûr d'avoir reçu le bon nombre de paramètre, puis d'utiliser `argv` pour accéder aux chaînes elles-mêmes. `argv[0]` étant le nom du programme, `argv[1]` est donc le premier paramètre passé au programme en ligne de commande. Il est important de noter que tous ces paramètres sont représentés comme des chaînes : si un paramètre de votre programme est censé être un nombre, vous devrez le convertir en nombre dans le programme à partir de la chaîne obtenue. Le Chapitre 21 vous apprendra à utiliser les conversions des bibliothèques standard. Le Listing 17.17 illustre l'utilisation des paramètres de la ligne de commande.

Listing 17.17 : Utilisation des paramètres de la ligne de commande

```
0:  //Listing 17.17 Paramètres de la ligne de commande
1:  #include <iostream>
2:  int main(int argc, char **argv)
3:  {
4:     std::cout << argc << " paramètres reçus...\n";
5:     for (int i = 0; i < argc; i++)
6:        std::cout << "paramètre "
6a:                  << i << " : " << argv[i] << std::endl;
7:     return 0;
8:  }
```

Ce programme produit le résultat suivant :

```
ProgTest je programme en C++
5 paramètres reçus...
paramètre 0 : ProgTest
paramètre 1 : je
paramètre 2 : programme
paramètre 3 : en
paramètre 4 : C++
```



Vous devez exécuter ce programme soit à partir de la ligne de commande (avec DOS), soit configurer les paramètres de ligne de commande de votre compilateur (voir la documentation de celui-ci).

La fonction `main()` déclare deux paramètre : `argc` est un entier qui contient le nombre de paramètres passés en la ligne de commande et `argv` est un pointeur sur un tableau de chaînes. Chaque chaîne du tableau pointé par `argv` est un paramètre de la ligne de commande. Notez que vous auriez pu déclarer ce paramètre comme `char* argv[]` ou comme `char argv[][]` : ce n'est qu'une question de style de programmation car, dans tous les cas, vous pourrez accéder aux différents paramètres en utilisant des indices sur ce tableau.

La ligne 4 utilise `argc` pour afficher le nombre de paramètres du programme : il y en a cinq en comptant le nom du programme.

Les lignes 5 et 6 affichent chaque paramètre en passant à `cout` les chaînes terminées par un caractère nul *via* un parcours du tableau des chaînes.

Une utilisation plus courante des options de la ligne de commande consiste à prendre un nom de fichier en paramètre, comme le montre le Listing 17.18.

Listing 17.18 : Utilisation des paramètres de la ligne de commande pour récupérer un nom de fichier

```
 0:  //Listing 17.18 Paramètres de la ligne de commande
 1:  #include <fstream>
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Animal
 6:  {
 7:  public:
 8:      Animal(int poids, long jours):
8a:       sonPoids(poids),saDureeVie(jours) {}
 9:      ~Animal(){}
10:
11:     int GetPoids()const { return sonPoids; }
12:     void SetPoids(int poids) { sonPoids = poids; }
13:
14:     long GetDureeVie()const { return saDureeVie; }
15:     void SetDureeVie (long jours) { saDureeVie = jours; }
16:
17:  private:
18:     int sonPoids;
19:     long saDureeVie;
20:  };
21:
22:  int main(int argc, char *argv[])   // renvoie 1 si erreur
23:  {
24:     if (argc != 2)
25:     {
```

```
26:        cout << "Usage : " << argv[0] << " <nom_fichier>\n";
27:        return(1);
28:     }
29:
30:     ofstream fout(argv[1], ios::binary);
31:     if (!fout)
32:     {
33:        cout << "Impossible d'ouvrir "
33a:            << argv[1] << " en écriture.\n";
34:        return(1);
35:     }
36:
37:     Animal Ours(50,100);
38:     fout.write((char*) &Ours,sizeof Ours);
39:
40:     fout.close();
41:
42:     ifstream fin(argv[1], ios::binary);
43:     if (!fin)
44:     {
45:        cout << "Impossible d'ouvrir "
45a:            << argv[1] << " en lecture.\n";
46:        return(1);
47:     }
48:
49:     Animal Ours2(1,1);
50:
51:     cout << "Poids Ours2 : " << Ours2.GetPoids() << endl;
52:     cout << "Jours Ours2 : " << Ours2.GetDaysAlive() << endl;
53:
54:     fin.read((char*) &Ours2, sizeof Ours2);
55:
56:     cout << "Poids Ours2 : " << Ours2.GetPoids() << endl;
57:     cout << "Jours Ours2 : " << Ours2.GetDureeVie() << endl;
58:     fin.close();
59:     return 0;
60:  }
```

Ce programme produit le résultat suivant :

```
Poids Ours2 : 1
Jours Ours2 : 1
Poids Ours2 : 50
Jours Ours2 : 100
```

La déclaration de la classe `Animal` est identique à celle du Listing 17.16 mais, au lieu de demander un nom de fichier à l'utilisateur, on utilise ici les paramètres de la ligne de

commande. `main()` est déclarée à la ligne 22 avec deux paramètres : le nombre de paramètres de la ligne de commande et un pointeur sur le tableau des chaînes de paramètres de la ligne de commande.

Les lignes 24 à 28 vérifient que `main()` a bien reçu deux paramètres ; dans le cas contraire, on affiche un message d'erreur :

```
Usage ProgTest <nom_fichier>
```

et le programme se termine. L'utilisation de `argv[0]` au lieu de mettre littéralement le nom du programme permet de compiler ce code afin de produire un programme portant n'importe quel nom : l'instruction `usage` affichera toujours le nom correct. Vous pouvez même renommer l'exécutable après sa compilation !

À la ligne 30, le programme tente d'ouvrir en écriture binaire le fichier dont le nom (`argv[1]`) est fourni en paramètre.

Cette technique est utilisée à la ligne 42 pour réouvrir le fichier en entrée, puis dans les instructions qui affichent un message d'erreur si le fichier ne peut pas être ouvert, aux lignes 33 et 45.

Questions-réponses

Q Comment savoir s'il faut utiliser les opérateurs d'insertion et d'extraction ou les autres fonctions membres de la classe `stream` ?

R Il est en général plus facile d'utiliser les opérateurs d'insertion et d'extraction et ils doivent être choisis de préférence lorsque leur comportement convient. Dans des circonstances plus inhabituelles, lorsque ces opérateurs ne conviennent plus (pour lire une suite de mots, par exemple), vous pouvez utiliser les autres méthodes.

Q Quelle différence y a-t-il entre `cerr` et `clog` ?

R `cerr` n'utilise pas de tampon. Tout ce qui est transmis vers `cerr` est immédiatement écrit. Ce comportement convient parfaitement aux erreurs qui s'affichent sur l'écran de la console, mais serait trop coûteux pour écrire ces erreurs sur disque. Les sorties de `clog`, par contre, passent par un tampon, ce qui est plus efficace, au risque toutefois de perdre une partie du journal si le programme plante.

Q Pourquoi a-t-on créé les flux puisque `printf()` fonctionne bien ?

R `printf()` ne sait pas gérer le typage fort de C++, ni les classes définies par l'utilisateur. La prise en charge de `printf()` n'est qu'un reste du langage C.

Q Quand doit-on utiliser `putback()` ?

R Lorsqu'une opération de lecture doit déterminer si un caractère est valide, mais qu'une autre opération de lecture (réalisée éventuellement par un autre objet) a besoin que ce caractère soit dans le tampon. Cette commande est généralement utilisée lors de l'analyse lexicale d'un fichier. Le compilateur C++, par exemple, pourrait utiliser `putback()`.

Q Peut-on utiliser `printf()` dans des programmes C++ ?

R Non, cette fonction doit à présent être considérée comme obsolète.

Testez vos connaissances

1. Qu'est-ce que l'opérateur d'insertion et à quoi sert-il ?
2. Qu'est-ce que l'opérateur d'extraction et à quoi sert-il ?
3. Quelles sont les trois formes de `get()` et quelles sont leurs différences ?
4. Quelle différence y a-t-il entre `cin.read()` et `cin.getline()` ?
5. Quelle est la largeur par défaut pour la sortie d'un entier `long` avec l'opérateur d'insertion ?
6. Quelle est la valeur renvoyée par l'opérateur d'insertion ?
7. Quel est le paramètre passé au constructeur d'un objet `ofstream` ?
8. Que fait l'argument `ios::ate` ?

Exercices

1. Écrivez un programme qui écrit dans les quatre objets `iostream` standard : `cin`, `cout`, `cerr` et `clog`.
2. Écrivez un programme qui demande à l'utilisateur d'entrer ses nom et prénom, puis qui les affiche à l'écran.
3. Modifiez le Listing 17.9 pour obtenir le même résultat sans utiliser `putback()` et `ignore()`.
4. Écrivez un programme qui reçoit un nom de fichier en paramètre et qui l'ouvre en lecture. Lisez chaque caractère du fichier et affichez uniquement les lettres et les caractères de ponctuations (ignorez tous les caractères non imprimables). Fermez le fichier avant la fin du programme.
5. Écrivez un programme qui affiche ses paramètres en ligne de commande dans l'ordre inverse, sans afficher le nom du programme.



18

Espaces de noms

Au sommaire de ce chapitre

- Résolution des fonctions et des classes par leurs noms
- Création d'un espace de noms
- Utilisation d'un espace de noms
- Utilisation de l'espace de noms standard `std`

Les espaces de noms aident les programmeurs à organiser les classes et à éviter les conflits de noms lors de l'utilisation de plusieurs bibliothèques.

Utilité des espaces de noms

Les conflits de noms sont un problème pour les programmeurs en C ou C++. Cette situation se produit lorsque deux noms identiques et de même portée sont utilisés à deux endroits d'un programme, ce qui est fréquent avec les bibliothèques. Une bibliothèque de classe conteneur, par exemple, a toutes les chances d'implémenter une classe `List`. (Les classes conteneur seront étudiées au Chapitre 19.)

Il n'est pas non plus surprenant de rencontrer une classe `List` dans une bibliothèque de fenêtrage pour stocker la liste des fenêtres ouvertes par une application. Supposons que vous utilisiez la classe `List` de la bibliothèque des classes conteneurs. Vous déclarez une instance de la classe `List` de la bibliothèque pour mémoriser vos fenêtres et vous découvrez que vous ne disposez pas des fonctions membres que vous vouliez appeler. Le compilateur a en fait associé votre déclaration au conteneur `List` de la bibliothèque standard alors que vous vouliez la classe `List` d'une bibliothèque spécifique.

Les espaces de noms permettent de réduire les conflits possibles. Ils ressemblent aux classes par certains côtés et leur syntaxe est d'ailleurs très similaire.

Les éléments déclarés au sein d'un espace de noms lui appartiennent et leur visibilité est publique. Les espaces de noms peuvent être imbriqués. Les fonctions peuvent être définies dans le corps de l'espace de noms ou à l'extérieur. Dans ce dernier cas, elles doivent être qualifiées à l'aide du nom de l'espace de noms ou le programme doit avoir importé l'espace de noms dans son espace global.

Résolution des fonctions et des classes par leurs noms

Lorsqu'il analyse le code source et construit une liste des noms de variables et de fonctions, le compilateur vérifie s'il existe des conflits de noms. Ceux qui ne peuvent être résolus par le compilateur le sont par l'éditeur de liens.

Le compilateur ne peut pas détecter les conflits de noms entre les fichiers objets ou autres unités de compilation ; c'est le rôle de l'éditeur de liens. Le compilateur n'affichera donc même pas le moindre message d'avertissement dans ce type de situation.

Il n'est pas rare que l'éditeur de liens affiche un message du type "*`Identificateur`* `multiply defined`" (où *`identificateur`* est un type nommé). Ce message signifie que vous avez utilisé le même nom avec la même portée dans des unités de compilation différentes. Vous obtiendrez par contre une erreur de compilation si cette redéfinition a lieu dans le même fichier. Les Listing 18.1a et 18.1b provoquent un message d'erreur de l'éditeur de liens.

Listing 18.1a : Premier listing utilisant la valeur *valeurEntiere*

```
0:  // fichier premier.cpp
1:  int valeurEntiere = 0 ;
2:  int main( ) {
3:      int valeurEntiere = 0 ;
4:      // . . .
5:      return 0 ;
6:  } ;
```

Listing 18.1b : Second listing utilisant la valeur *valeurEntiere*

```
0:  // fichier second.cpp
1:  int valeurEntiere = 0 ;
2: // fin de second.cpp
```

Le message de l'éditeur de liens sur notre système est : "`in second.obj: valeurEntiere already defined in premier.obj`" (c'est-à-dire "*dans second.obj : valeurEntiere* est *déjà définie dans premier.obj*"). Ce message n'apparaîtrait pas si les noms avaient une portée différente.

Le compilateur peut également envoyer un avertissement du type *`identificateur`* `hiding`, signalant que `valeurEntiere` dans `main()` *masque* la variable globale de même nom.

Pour utiliser la variable `valeurEntiere` déclarée à l'extérieur de `main()`, vous devez explicitement lui attribuer une portée globale. L'exemple des Listings 18.2a et 18.2b affecte la valeur `10` à la variable `valeurEntiere` externe à `main()`, pas à celle qui est déclarée dans `main()`.

Listing 18.2a : Premier listing pour le masquage de l'identificateur

```
0:  // fichier premier.cpp
1:  int valeurEntiere = 0 ;
2:  int main( )
3:  {
4:      int valeurEntiere = 0 ;
5:      ::valeurEntiere = 10 ; // valeurEntiere globale
6:      // . . .
7:      return 0 ;
8:  } ;
```

Listing 18.2b : Second listing pour le masquage de l'identifiant

```
0:  // fichier second.cpp
1:  int valeurEntiere = 0 ;
2:  // fin de second.cpp
```



Notez l'usage de l'opérateur de résolution de portée `::` qui indique que la variable `valeurEntiere` concernée est globale et non locale.

Le problème avec ces deux listings est qu'ils définissent chacun une variable globale qui a le même nom et la même visibilité, ce qui provoquera une erreur lors de l'édition des liens.

Visibilité des variables

Le terme *visibilité* désigne la portée d'un objet défini, qu'il s'agisse d'une variable, d'une classe ou d'une fonction. Bien que nous ayons déjà évoqué ce problème au Chapitre 5, il mérite d'être revu brièvement ici.

Une variable déclarée et définie en dehors de toute fonction a une portée *globale* ou au niveau *fichier*. Sa visibilité s'étend du point de sa définition jusqu'à la fin du fichier. Une variable dont la portée est *locale* ou au niveau du *bloc* se trouve dans une structure d'un bloc. Ce sont généralement des variables définies dans les fonctions. Le Listing 18.3 montre des exemples de portées.

Listing 18.3 : Portées des variables

```
 0:  // Listing 18.3
 1:  int intPorteeGlobale  = 5 ;
 2:  void f( )
 3:  {
 4:      int intPorteeLocale = 10 ;
 5:  }
 6:  int main( )
 7:  {
 8:      int intPorteeLocale = 15 ;
 9:      {
10:         int autreLocale = 20 ;
11:         int intPorteeLocale = 30 ;
12:     }
13:     return 0 ;
14: }
```

La première définition, `intPorteeGlobale`, est visible dans les fonctions `f()` et `main()`. La définition suivante, `intPorteeLocale`, se trouve dans la fonction `f()` à la ligne 4. La portée de cette variable est locale, elle n'est donc visible que dans le bloc qui la définit.

La fonction `main()` ne peut accéder à la variable `intPorteeLocale` de `f()`, puisqu'elle est devenue hors de portée au retour de la fonction. La troisième définition, également appelée `intPorteeLocale`, se trouve à la ligne 8, dans la fonction `main()` ; sa portée est limitée au niveau du bloc.

La variable `intPorteeLocale` de `main()` n'est pas en conflit avec celle de `f()`. Les deux définitions qui suivent, lignes 10 et 11, `autreLocale` et `intPorteeLocale`, ont une portée

au niveau du bloc. Dès qu'elles atteignent l'accolade fermante de la ligne 12, elles deviennent hors de portée.

Cette dernière variable `intPorteeLocale` masque celle de même nom qui a été définie juste avant l'accolade ouvrante (la seconde `intPorteeLocale` définie dans le programme). Lorsque le programme dépasse l'accolade fermante de la ligne 12, la seconde variable `intPorteeLocale` récupère sa visibilité. Tout changement apporté à la variable `intPortee-Locale` définie entre les accolades n'affecte pas la variable externe de même nom.

Liaison

Les noms peuvent avoir une liaison *interne* et *externe*. Ces deux termes désignent l'utilisation ou la disponibilité d'un nom dans plusieurs unités de compilation ou au sein d'une seule unité. Un nom ayant une liaison interne ne peut être référencé qu'au sein de l'unité dans laquelle il est défini. Par exemple, une variable définie comme ayant une liaison interne peut être partagée par les fonctions au sein de la même unité, alors que les variables ayant une liaison externe sont également accessibles aux autres unités. Les Listings 18.4a et 18.4b montrent ces deux types de liaisons.

Listing 18.4a : Liaisons interne et externe

```
0:  // fichier : premier.cpp
1:  int intExterne = 5 ;
2:  const int j = 10 ;
3:  int main()
4:  {
5:      return 0 ;
6:  }
```

Listing 18.4b : Liaisons interne et externe

```
0:  // fichier : second.cpp
1:  extern int intExterne;
2:  int unIntExterne = 10 ;
3:  const int j = 10 ;
```

La variable `intExterne` définie à la ligne 1 de `premier.cpp` (Listing 18.4a) a une liaison externe. Bien qu'elle soit définie dans `premier.cpp`, `second.cpp` peut donc aussi y avoir accès. Les deux `j` présentes dans les deux fichiers sont des constantes qui, par défaut, ont une liaison interne. Vous pouvez redéfinir cette propriété des `const` en fournissant une déclaration explicite, comme le montrent les Listing 18.5a et 18.5b.

Listing 18.5a : Redéfinir la liaison par défaut de `const`

```
0:  // fichier: premier.cpp
1:  extern const int j = 10 ;
```

Listing 18.5b : Redéfinir la liaison par défaut de `const`

```
0:  // fichier: second.cpp
1:  extern const int j ;
2:  #include <iostream>
3:  int main()
4:  {
5:      std::cout << "j vaut " << j << std::endl ;
6:      return 0 ;
7:  }
```

Remarquez l'usage de `cout` à la ligne 5 en précisant explicitement l'espace de nom `std`. Cet exemple affiche le résultat suivant :

```
j vaut 10
```

Variables statiques globales

Les variables statiques globales sont désormais considérées comme obsolètes. Elles sont déclarées de la façon suivante :

```
static int intStatique = 10 ;
int main()
{
    //...
}
```

L'utilisation de `static` pour limiter la portée des variables externes n'est plus recommandée et peut même devenir illégale à l'avenir. Il faut désormais utiliser les espaces de noms.

Faire	*Ne pas faire*
● Utiliser les espaces de noms au lieu des variables statiques globales.	● Appliquer le mot-clé `static` à une variable définie au niveau du fichier.

Création d'un espace de noms

La syntaxe de la déclaration d'un espace de noms ressemble à celle d'une déclaration de classe ou de structure. On commence par le mot-clé `namespace`, suivi éventuellement d'un nom d'espace de noms, puis d'une accolade ouvrante. L'espace de noms se termine par une accolade fermante, sans point-virgule.

Exemple :

```
namespace Fenetre
{
    void deplacer( int x, int y) ;

    class Liste
    {
        // ...
    }

}
```

Le nom `Fenetre` identifie l'espace de noms de façon unique. Plusieurs occurrences d'un même nom d'espace de noms peuvent exister dans le même fichier ou dans plusieurs unités de compilation. Dans ce cas, les différentes instances seront fusionnées par le compilateur afin de former un seul espace de noms. C'est d'ailleurs ce qui se passe pour l'espace de noms de la bibliothèque C++ Standard, `std` : la bibliothèque standard est un regroupement logique de plusieurs fonctionnalités, mais elle est trop volumineuse et trop complexe pour être stockée dans un seul fichier.

Le concept essentiel des espaces de noms est de regrouper les éléments connexes dans un emplacement (nommé) spécifié. Les Listing 18.6a et 18.6b en montrent des exemples.

Listing 18.6a : Regrouper des éléments connexes

```
0:  // entete1.h
1:  namespace Fenetre
2:  {
3:      void deplacer( int x, int y) ;
4:  }
```

Listing 18.6b : Regrouper des éléments connexes

```
0:  // entete2.h
1:  namespace Fenetre
2:  {
3:      void modifTaille( int x, int y ) ;
4:  }
```

Comme vous pouvez le constater, l'espace de noms `Fenetre` est réparti sur les deux fichiers d'en-tête. Le compilateur considère les fonctions `deplacer()` et `modifTaille()` comme faisant partie de l'espace de noms `Fenetre`.

Déclaration et définition de types

Les types et les fonctions peuvent être déclarés et définis dans des espaces de noms. C'est, bien sûr, un choix de conception et de maintenance. Nous avons déjà précisé qu'il était toujours préférable de séparer l'interface de l'implémentation ; ce principe vaut aussi bien pour les classes que pour les espaces de noms. L'extrait de code suivant illustre un mauvais exemple de définition d'espace de noms :

```
namespace Fenetre {
    // . . . autres déclarations et définitions de variable.
    void deplacer( int x, int y) ; // déclarations
    void modifTaille( int x, int y ) ;
    // . . . autres déclarations et définitions de variable.

    void deplacer( int x, int y )
    {
        if( x < MAX_SCREEN_X  && x > 0 )
            if( y < MAX_SCREEN_Y  && y > 0 )
                platforme.move( x, y ) ; // routine spécifique
    }

    void modifTaille( int x, int y )
    {
        if( x < MAX_SIZE_X  && x > 0 )
            if( y < MAX_SIZE_Y  && y > 0 )
                platforme.resize( x, y ) ; // routine spécifique
    }
    // . . . suite des définitions
}
```

Un espace de noms comme celui-ci devient très rapidement confus ! Cet exemple fait à peu près 20 lignes ; imaginez si l'espace de noms était quatre fois plus grand !

Définition de fonctions en dehors d'un espace de noms

Les fonctions doivent être définies en dehors de l'espace de noms. Cette façon de procéder permet de séparer clairement la déclaration de la fonction de sa définition et d'aérer le corps de l'espace de noms. En outre, cela permet de placer l'espace de noms et ses déclarations dans un fichier d'en-tête, alors que les définitions seront dans un fichier d'implémentation. Les Listings 18.7a et 18.7b illustrent cette séparation.

Listing 18.7a : Déclarer un en-tête dans un espace de nom

```
0:  // fichier entete.h
1:  namespace Fenetre {
2:      void deplacer( int x, int y) ;
3:      // autres déclarations ...
4:  }
```

Listing 18.7b : Déclarer l'implémentation dans le fichier source

```
0:  // fichier impl.cpp
1:  void Fenetre::deplacer( int x, int y )
2:  {
3:      // code pour le déplacement de la fenêtre
4:  }
```

Ajout de nouveaux membres

Vous ne pouvez ajouter de nouveaux membres que dans le corps de l'espace de noms. Il est impossible de les ajouter en utilisant une syntaxe avec qualification car vous obtiendriez un message de votre compilateur, comme dans l'exemple suivant :

```
namespace Fenetre {
    // quantité de déclarations
}
//...du code
int Fenetre::nouveauEntierDansEspaceDeNoms ; // Interdit !!!
```

Cette ligne de code est illégale. La solution est de déplacer la déclaration dans le corps de l'espace de noms.

Lorsque vous ajoutez de nouveaux membres, il ne faut pas ajouter de modificateurs d'accès, comme `private` ou `public`. Tous les membres au sein d'un espace de noms sont publics. Le code suivant ne pourra donc pas être compilé, car vous ne pouvez pas utiliser `private` :

```
namespace Fenetre {
    private:
        void deplacer( int x, int y ) ;
}
```

Imbriquer les espaces de noms

La définition d'un espace de noms étant aussi une déclaration, vous pouvez les imbriquer. Comme pour les autres espaces de noms, vous devez qualifier un nom par celui de l'espace

de noms qui l'englobe. Si vous avez imbriqué des espaces de noms, leurs noms devront donc tous apparaître dans la qualification. L'exemple suivant montre un espace de noms imbriqué dans un autre espace de noms :

```
namespace Fenetre {
    namespace Panneau {
        void taille( int x, int y ) ;
    }
}
```

Pour accéder à la fonction `taille()` en dehors de l'espace de nom `Fenetre`, vous devez qualifier la fonction avec les deux noms des espaces de noms. Vous devrez donc utiliser la syntaxe suivante :

```
Fenetre::Panneau::taille( 10, 20 ) ;
```

Utilisation d'un espace de noms

Examinons un exemple d'utilisation d'un espace de noms avec l'opérateur de résolution de portée. Ici, nous déclarerons tous les types et les fonctions qui doivent être employés dans l'espace de nom `Fenetre`. Après la définition de tout ce qui est requis, toutes les fonctions membres qui ont été déclarées seront définies à l'extérieur de l'espace de noms ; les noms sont identifiés explicitement à l'aide de l'opérateur de résolution de portée. Reportez-vous au Listing 18.8.

Listing 18.8 : Utilisation d'un espace de noms

```
 0: #include <iostream>
 1: // Utilisation d'un espace de noms
 2:
 3: namespace Fenetre
 4: {
 5:     const int MAX_X = 30 ;
 6:     const int MAX_Y = 40 ;
 7:     class Panneau
 8:     {
 9:         public:
10:             Panneau() ;
11:             ~Panneau() ;
12:             void taille( int x, int y ) ;
13:             void deplacer( int x, int y ) ;
14:             void afficher( ) ;
15:         private:
16:             static int cpteur;
```

```
17:             int x ;
18:             int y ;
19:     };
20: }
21:
22: int Fenetre::Panneau::cpteur = 0 ;
23: Fenetre::Panneau::Panneau() : x(0), y(0) { }
24: Fenetre::Panneau::~Panneau() { }
25:
26: void Fenetre::Panneau::taille( int x, int y )
27: {
28:     if( x < Fenetre::MAX_X  &&  x > 0 )
29:         Panneau::x = x ;
30:     if( y < Fenetre::MAX_Y  &&  y > 0 )
31:         Panneau::y = y ;
32: }
33: void Fenetre::Panneau::deplacer( int x, int y )
34: {
35:     if( x < Fenetre::MAX_X  &&  x > 0 )
36:         Panneau::x = x ;
37:     if( y < Fenetre::MAX_Y  &&  y > 0 )
38:         Panneau::y = y ;
39: }
40: void Fenetre::Panneau::afficher( )
41: {
42:     std::cout << "x " << Panneau::x ;
43:     std::cout << " y " << Panneau::y << std::endl ;
44: }
45:
46: int main( )
47: {
48:     Fenetre::Panneau panneau ;
49:
50:     panneau.deplacer( 20, 20 ) ;
51:     panneau.afficher( ) ;
52:
53:     return 0 ;
54: }
```

Ce programme produit le résultat suivant :

```
x 20 y 20
```

Vous remarquerez que la classe `Panneau` (lignes 7 à 19) est imbriquée dans l'espace de noms `Fenetre` (lignes 3 à 20). C'est la raison pour laquelle vous devez qualifier le nom `Panneau` avec le qualificateur `Fenetre::`.

La variable statique `cpteur`, déclarée dans `Panneau` à la ligne 16, est définie de la façon habituelle. Notez que dans la méthode `Panneau::taille()`, `MAX_X` et `MAX_Y` sont qualifiés aux lignes 26 à 32 pour éviter une erreur de compilation. Ceci est également valable pour la méthode `Panneau::deplacer()`.

Notez également la qualification de `Panneau::x` et `Panneau::y` au sein des deux définitions de méthodes. En effet, si la fonction `Panneau::deplacer()` était écrite de la façon suivante, vous auriez un souci :

```
void Fenetre::Panneau::deplacer( int x, int y )
{
    if( x < Fenetre::MAX_X  &&  x > 0 )
        x = x ;
    if( y < Fenetre::MAX_Y  &&  y > 0 )
        y = y ;
    Platforme::deplacer( x, y ) ;
}
```

Voyez-vous où se situe le problème ? Le message de votre compilateur ne vous sera pas d'un grand secours, certains n'émettront même aucun diagnostic.

La source du problème réside dans les paramètres de la fonction : `x` et `y` masquent les variables d'instance privées `x` et `y` de la classe `Panneau`. En réalité, les instructions suivantes affecteraient `x` et `y` à elles-mêmes :

```
x = x ;
y = y ;
```

Mot-clé *using*

Le mot-clé `using` est utilisé à la fois pour la *directive* `using` et la *déclaration* `using`. C'est la syntaxe employée qui permettra de déterminer s'il s'agit d'une directive ou d'une déclaration.

La directive *using*

La directive `using` expose dans la portée courante tous les noms déclarés dans un espace de noms. Vous pouvez ensuite faire référence à ces noms sans avoir besoin de les qualifier à l'aide de leurs espaces de noms respectifs. Par exemple :

```
namespace Fenetre {
    int valeur1 = 20 ;
    int valeur2 = 40 ;
}
```

```
. . .
Fenetre::valeur1 = 10 ;

using namespace Fenetre ;
valeur2 = 30 ;
```

La portée de la directive `using` s'étend de sa déclaration jusqu'à la fin de la portée courante. Remarquez que `valeur1` doit être qualifiée pour pouvoir être référencée, contrairement à la `valeur2` car la directive `using` a introduit dans la portée courante tous les noms de l'espace de noms.

La directive `using` peut être utilisée à tous les niveaux de portée, ce qui permet de l'employer notamment dans la portée d'un bloc ; lorsque ce bloc devient hors de portée, tous les noms de l'espace de noms disparaissent également. Examinez l'exemple suivant :

```
namespace Fenetre {
    int valeur1 = 20 ;
    int valeur2 = 40 ;
}
//. . .
void f()
{
    {
        using namespace Fenetre ;
        valeur2 = 30 ;
    }
    valeur2 = 20 ; // Erreur !
}
```

La dernière ligne de code provoque une erreur car `valeur2` n'est pas définie. Ce nom est accessible dans le bloc précédent puisque la directive l'a intégré dans ce bloc. Lorsque ce dernier devient hors de portée, les noms de l'espace de noms `Fenetre` le deviennent également. Pour que `valeur2` fonctionne à la dernière ligne, vous devez la qualifier totalement :

```
Fenetre::valeur2 = 20;
```

Les noms de variable déclarés dans une portée locale masquent tous les noms de l'espace de nom introduits dans cette portée. Ce comportement est similaire à la façon dont une variable locale masque une variable globale. Même si vous introduisez un espace de noms après une variable locale, celle-ci masquerait quand même le nom de l'espace de noms. Exemple :

```
namespace Fenetre {
    int valeur1 = 20 ;
    int valeur2 = 40 ;
}
//. . .
```

```
void f()
{
   int valeur2 = 10 ;
   using namespace Fenetre ;
   std::cout << valeur2 << std::endl  ;
}
```

Cette fonction affichera `10` et non `40`. La variable `valeur2` de l'espace de noms `Fenetre` est masquée par la variable `value2` locale à `f()`. Si vous voulez utiliser un nom d'un espace de noms, vous devez le qualifier totalement.

Une ambiguïté peut se produire lors de l'utilisation d'un nom défini à la fois globalement et dans un espace de noms. Cette ambiguïté n'apparaît que si le nom est employé, pas lors de l'introduction de l'espace de noms. Le fragment de code suivant en fait la démonstration :

```
namespace Fenetre {
    int valeur1 = 20 ;
}
//. . .
using namespace Fenetre ;
int valeur1 = 10 ;
void f( )
{
   valeur1 = 10 ;
}
```

L'ambiguïté intervient dans la fonction `f()`. La directive introduit effectivement `Fenetre::valeur1` dans l'espace de noms global ; `valeur1` étant déjà définie globalement, l'emploi de `value1` dans `f()` est donc une erreur. Notez que si vous supprimez la ligne de code dans `f()`, aucune erreur n'apparaîtra.

La déclaration *using*

Une déclaration `using` ressemble à une directive `using`, sauf qu'une déclaration permet un niveau de contrôle plus fin. Plus précisément, une déclaration `using` sert à déclarer qu'un nom spécifique (provenant d'un espace de noms) fait désormais partie de la portée courante. Vous pouvez ensuite faire référence à l'objet concerné par son nom seul. L'exemple suivant montre comment utiliser une déclaration `using` :

```
namespace Fenetre {
    int valeur1 = 20 ;
    int valeur2 = 40 ;
    int valeur3 = 60 ;
}
//. . .
```

```
using Fenetre::valeur2; // met valeur2 dans la portée courante
Fenetre::valeur1 = 10 ; // valeur1 doit être qualifiée
valeur2 = 30 ;
Fenetre::valeur3 = 10 ; // valeur3 doit être qualifiée
```

La déclaration `using` ajoute le nom indiqué à la portée courante ; elle n'affecte pas les autres noms de l'espace de noms. Dans l'exemple précédent, `valeur2` est utilisée sans qualification, mais `valeur1` et `valeur3` requièrent une qualification. Contrairement à la directive, qui ajoute tous les noms d'un espace de noms à la portée courante, une déclaration permet donc de choisir ceux que l'on veut ajouter.

Lorsqu'un nom a été ajouté à une portée, il reste visible jusqu'à la fin de celle-ci, comme pour toute déclaration. Une déclaration `using` peut être utilisée dans l'espace de noms global ou dans n'importe quel portée locale.

Il est interdit d'introduire un nom d'espace de noms existant déjà dans une portée locale. L'inverse est également vrai. L'exemple suivant en fait la démonstration :

```
namespace Fenetre {
    int valeur1 = 20 ;
    int valeur2 = 40 ;
}
//. . .
void f()
{
   int valeur2 = 10 ;
   using Fenetre::valeur2; // déclaration dupliquée
   std::cout << valeur2 << std::endl  ;
}
```

La deuxième ligne de la fonction `f()` provoquera une erreur de compilation, car le nom `valeur2` est déjà défini. La même erreur se serait produite si la déclaration `using` avait été placée avant la définition de la variable locale `valeur2`.

Tout nom introduit dans une portée locale à l'aide d'une déclaration `using` masque tout nom identique à l'extérieur de cette portée, comme le montre l'exemple suivant :

```
namespace Fenetre {
    int valeur1 = 20 ;
    int valeur2 = 40 ;
}
int valeur2 = 10 ;
//. . .
void f()
{
    using Fenetre::valeur2;
    std::cout << valeur2 << std::endl  ;
}
```

La déclaration `using` de `f()` masque la variable `valeur2` définie dans l'espace de noms global.

Comme on l'a déjà indiqué, une *déclaration* `using` permet un contrôle plus fin sur les noms introduits à partir d'un espace de noms. Une *directive* `using` ajoute tous les noms d'un espace de noms à la portée courante. Il est donc préférable d'utiliser une déclaration car une directive va, en fait, à l'encontre de l'intérêt du mécanisme des espaces de noms. Une déclaration est plus définitive car on identifie explicitement tous les noms que l'on souhaite introduire dans une portée. Elle ne risque donc pas de polluer l'espace de nom global comme une directive (sauf si, bien sûr, vous déclarez tous les noms de l'espace de noms). Avec une déclaration `using`, le masquage de nom, la pollution de l'espace de noms global et les ambiguïtés sont réduits à un niveau plus raisonnable.

Alias d'espace de noms

Un *alias* d'espace de noms permet d'attribuer un autre nom à un espace de noms existant et permet ainsi de créer un raccourci pour faire référence à cet espace de nom. L'intérêt est évident lorsque le nom de l'espace de noms est particulièrement long. Examinez le code suivant :

```
namespace la_compagnie_informatique {
    int valeur;
    // . . .
}
la_compagnie_informatique::valeur = 10 ;
. . .
namespace LCI = la_compagnie_informatique ;
TSC::valeur = 20 ;
```

Le risque, bien entendu, est d'entrer un conflit avec un nom existant. Dans ce cas, le compilateur le signalera.

Espace de noms anonyme

Un espace de noms anonyme est simplement un espace de noms sans nom. On les utilise souvent pour protéger les variables globales des conflits de noms potentiels entre les fichiers objet et les unités de compilation. Chaque unité de compilation a son propre espace de noms anonyme, qui lui est unique. Tous les noms définis dans l'espace de nom anonyme (au sein de chaque unité) sont accessibles sans qualification explicite.

Les Listings 18.9a et 18.9b illustrent deux espaces de noms anonymes dans deux fichiers séparés.

Listing 18.9a : Un espace de nom anonyme

```
0:  // fichier: un.cpp
1:  namespace {
2:  {
3:     int valeur;
4:     char p( char *p ) ;
5:     //. . .
6:  }
```

Listing 18.9b : Un second espace de nom anonyme

```
 0:  // fichier: deux.cpp
 1:  namespace
 2:  {
 3:      int valeur;
 4:      char p( char *p ) ;
 5:      //. . .
 6:  }
 7:  int main( )
 8:  {
 9:      char c = p( ptr ) ;
10: }
```

Si ces deux listings étaient compilés dans le même exécutable, chacun des noms `valeur` et `p()` serait distinct pour son fichier respectif. Pour accéder à un nom (d'espace de noms anonyme) dans une unité, on utilise ce nom sans qualification. C'est ce que l'on constate dans l'exemple précédent avec l'appel à la fonction `p()` dans chaque fichier.

Cette utilisation implique une directive `using` pour les objets référencés à partir d'un espace de noms anonyme. Il en résulte que vous ne pouvez pas accéder aux membres d'un espace de noms anonyme dans une autre unité.

Le comportement d'un espace de noms anonyme est donc comparable à celui d'un objet statique ayant une liaison externe. Considérez l'exemple suivant :

```
static int valeur = 10 ;
```

Souvenez-vous cependant que cet usage du mot-clé `static` est maintenant déconseillé au profit des espaces de noms. On peut aussi comparer les espaces de noms anonymes à des variables globales avec liaison interne.

L'espace de noms standard *std*

Le meilleur exemple d'espace de noms se trouve dans la bibliothèque C++ standard. Celle-ci est, en effet, complètement contenue dans un espace de noms appelé `std`. Toutes les fonctions, classes, objets et modèles sont déclarés dans cet espace de noms.

Vous avez déjà rencontré du code comme celui-ci :

```
#include <iostream>
using namespace std ;
```

Vous savez maintenant que l'utilisation de cette directive `using` place tout le contenu de `std` dans l'espace de noms global.

Il n'est pas souhaitable d'employer la directive `using` avec la bibliothèque standard. Pourquoi ? Parce que cela pollue l'espace de noms global de vos applications avec tous les noms trouvés dans l'en-tête. Souvenez-vous que tous les fichiers d'en-tête utilisent cet espace de noms ; si vous incluez plusieurs fichiers d'en-tête standard et que vous utilisez la directive `using`, tout ce qui est déclaré dans ces en-têtes se retrouvera dans l'espace de noms global.

Vous aurez d'ailleurs peut-être remarqué que la plupart des exemples de ce livre violent cette règle, mais c'est uniquement pour la brièveté des exemples. Vous devriez plutôt utiliser la déclaration `using`, comme dans le Listing 18.10.

Listing 18.10 : La méthode correcte pour utiliser les éléments d'espace de nom `std`

```
 0:  #include <iostream>
 1:  using std::cin ;
 2:  using std::cout ;
 3:  using std::endl ;
 4:  int main( )
 5:  {
 6:     int valeur = 0 ;
 7:     cout << "Combien d'oeufs voulez-vous ?" << endl ;
 8:     cin >> valeur;
 9:     cout << valeur << " oeufs au plat !" << endl ;
10:     return( 0 ) ;
11: }
```

Ce programme produit le résultat suivant :

```
Combien d'oeufs voulez-vous ?
4
4 oeufs, au plat !
```

Comme vous pouvez le constater, nous utilisons trois éléments de l'espace de noms `std`. Ils sont déclarés des lignes 1 à 3.

Vous pouvez aussi qualifier pleinement les noms que vous utilisez, comme dans le Listing 18.11.

Listing 18.11 : Qualifier en ligne les éléments des espaces de noms

```
 0:  #include <iostream>
 1:  int main( )
 2:  {
 3:      int valeur = 0 ;
 4:      std::cout << "Combien d'oeufs voulez-vous ?"
4a:                << std::endl ;
 5:      std::cin >> valeur;
 6:      std::cout << valeur << " oeufs au plat !" << std::endl ;
 7:      return( 0 ) ;
 8:  }
```

Ce programme produit le résultat suivant :

```
Combien d'oeufs voulez-vous ?
4
4 oeufs au plat !
```

Qualifier en ligne les éléments des espaces de nom est justifié dans le cas de programmes courts, mais peut vite devenir fastidieux si le code devient plus important. Imaginez l'ennui d'avoir à préfixer par `std::` chaque nom provenant de la bibliothèque standard !

Questions-réponses

Q Est-il obligatoire d'avoir recours aux espaces de noms ?

R Non : pour des programmes simples, vous pouvez vous passer des espaces de noms. Vérifiez que vous utilisez les anciennes bibliothèques standard (par exemple, `#include <string.h>`) plutôt que les nouvelles (comme `#include <cstring>`). Cependant, pour toutes les raisons que nous avons expliquées dans ce chapitre, il est préférable d'utiliser les espaces de noms.

Q C++ est-il le seul langage à utiliser les espaces de noms ?

R Non. D'autres langages les emploient aussi pour organiser et séparer les valeurs : Visual Basic 7 (.NET), C#, etc. D'autres langages disposent de concepts similaires. Java, par exemple, utilise des paquetages.

Q Qu'est-ce qu'un espace de noms anonyme ? Quelle est son utilité ?

R C'est un espace de noms qui n'a pas de nom. Il sert à envelopper une collection de déclarations afin d'éviter d'éventuels conflits de noms. Les noms dans un espace de noms anonyme ne sont pas utilisables en dehors de l'unité où il est déclaré.

Testez vos connaissances

1. Comment accéder à la fonction `MaFonction()` si elle se trouve dans l'espace de noms `Interne` de l'espace de nom `Externe` ?

```
Externe::Interne::MaFonction();
```

2. Étudiez le code suivant :

```
int x = 4;
int main()
{
   for( y = 1; y < 10; y++)
   {
       cout << y << ":" << endl;
       {
           int x = 10 * y;
           cout << "X = " << x << endl
       }
   }
   // *** ICI ***
}
```

   Quelle est la valeur de `X` lorsque ce programme atteind "ICI" dans ce listing ?

3. Est-il possible d'utiliser des noms définis dans un espace de noms sans avoir recours au mot-clé `using` ?
4. Quelles sont les principales différences entre un espace de nom normal et anonyme ?
5. Quelles sont les deux formes d'instructions utilisant le mot-clé `using` ? Quelles sont les différences entre les deux ?
6. Que sont les espaces de noms anonymes ? À quoi servent-ils ?
7. Qu'est-ce que l'espace de noms standard ?

Exercices

1. CHERCHEZ L'ERREUR ! Ce programme est bogué :

```
0:  #include <iostream>
1:  int main()
2:  {
3:      cout << "Bonjour !" << end;
4:      return 0;
5:  }
```

2. Proposez trois solutions pour résoudre le problème du programme précédent.
3. Écrivez le code permettant de déclarer un espace de noms `MesElements`. Cet espace de noms doit contenir une classe appelée `MaClasse`.



19

Les modèles

Au sommaire de ce chapitre

- Les modèles et leur utilisation
- La création de modèles de classes
- La création de modèles de fonctions
- La bibliothèque des modèles standard et certains de ses modèles

Les types paramétrés ou modèles constituent un outil très puissant pour les programmeurs C++. L'importance des modèles ayant été reconnue, la bibliothèque des modèles standard (STL, *Standard Template Library*) a été intégrée à la définition du langage C++.

Qu'est-ce qu'un modèle

À la fin de la deuxième partie, vous avez appris à construire un objet `ListePieces` et à l'utiliser pour créer un `Catalogue`. Si vous voulez vous appuyer sur l'objet `ListePieces` pour une liste de chats, vous rencontrerez un problème : `ListePieces` ne connaît que des pièces.

Pour résoudre ce problème, vous pouvez créer une classe de base `Liste`. Vous pourrez ainsi couper et coller entre la classe `ListePieces` et la nouvelle déclaration `ListeChats`. Plus tard, lorsque vous aurez besoin d'une liste d'objets `Voiture`, vous devrez créer une nouvelle classe `ListeVoitures` et recommencer le copier/coller.

Inutile de préciser qu'il ne s'agit pas d'une solution satisfaisante. Au cours du temps, la classe `Liste` et ses classes dérivées devront sûrement être étendues. S'assurer que toutes les modifications nécessaires sont propagées dans toutes les classes concernées risque de devenir rapidement un cauchemar.

Vous pourriez aussi faire hériter `Chat` de `Pieces` pour qu'un chat puisse entrer dans la hiérarchie d'héritage et qu'une collection de pièces puisse aussi contenir des chats. Cela pose évidemment un problème quant à la propreté de la hiérarchie conceptuelle des classes, car les chats ne sont pas des pièces.

Il serait également possible de créer une classe `Liste` qui contiendrait des objets d'une classe `"Object"` et qui pourrait donc contenir tous les objets dérivant de cette classe de base. Cependant, cela diminuerait le typage fort et compliquerait la tâche du compilateur pour vérifier que votre programme est correct. Vous voulez en fait créer une famille de classes apparentées, dont la seule différence sera le type des éléments sur lequel elles opèrent et vous souhaitez n'effectuer les modifications qu'à un seul endroit afin de minimiser vos opérations de maintenance.

La création et l'utilisation de modèles représentent une solution à ce problème. Bien qu'à l'origine ils étaient absents du langage C++, ils font maintenant partie intégrante de la norme. Comme tous les éléments de C++, ils sont souples et fournissent une gestion robuste des types, mais ils peuvent toutefois rebuter les débutants. Cependant, lorsque aurez compris leur fonctionnement, vous constaterez qu'ils sont très utiles.

Les modèles permettent de créer une classe dont le type des éléments qu'elle manipule peut être modifié. Vous pouvez, par exemple, les utiliser pour indiquer au compilateur comment fabriquer une liste de n'importe quel type d'objet au lieu de créer un ensemble de listes spécifiques – `ListePieces` pour une liste de pièces, `ListeChats` pour une liste de chats. En effet, le seul point qui différencie ces deux listes est le *type* de leurs éléments. Avec les modèles, ce type devient un paramètre de la définition de la classe. Vous pouvez donc créer une famille de classes à partir du modèle, chacune étant configurée pour fonctionner avec un type d'élément différent.

Les bibliothèques C++ contiennent très souvent une classe tableau. Mais créer une classe de tableaux pour les entiers, une autre pour les `double` et une troisième pour des animaux est fastidieux et inefficace. Les modèles permettent de déclarer une seule classe de tableau paramétrée, puis de préciser le type des éléments de chaque *instance* du tableau.

Bien que la bibliothèque des modèles standard fournit un jeu de classes *conteneurs* standardisé contenant des tableaux, des listes et ainsi de suite, le meilleur moyen d'apprendre les modèles consiste à créer les siens ! Dans les sections qui suivent, nous allons donc créer nos propres classes conteneurs pour bien comprendre comment fonctionnent les modèles. Dans un contexte commercial, par contre, vous aurez tout intérêt à utiliser les classes de la STL au lieu de créer les vôtres.

La création d'un type spécifique à partir d'un modèle est appelée *instanciation*, les classes individuelles sont alors appelées instances du modèle.



Les instances d'un modèle diffèrent des instances d'objets créées à l'aide du modèle. Plus généralement, "l'instanciation" sert à désigner la création d'une instance (objet) à partir d'une classe. Il faut donc que le contexte d'utilisation du mot "instanciation" soit bien clair.

Les *modèles paramétrés* permettent de créer une classe générale et de lui passer des types en paramètres pour construire des instances spécifiques.

Avant de pouvoir instancier un modèle, vous devez toutefois le définir.

Construire une définition de modèle

La déclaration de base d'un modèle débute par le mot-clé `template`, suivi d'un paramètre pour le type. Le format est le suivant :

```
template <class T>
```

Ici, `template` et `class` sont des mots-clés. `T` est un emplacement – un peu comme un nom de variable. Cet emplacement pourrait porter n'importe quel nom mais on utilise généralement `T` ou `Type`. La valeur de `T` sera un type de données.

Le mot-clé `class` pouvant être confus dans ce contexte, vous pouvez utiliser le mot-clé `typename` à la place :

```
template <typename T>
```

Dans ce chapitre, nous utiliserons le plus souvent la mot-clé `class`. Le mot-clé `typename` indique toutefois plus clairement ce que vous définissez lorsque le paramètre du modèle est un type primitif et non une classe.

Si l'on revient à la création de notre propre type tableau, nous pouvons donc utiliser l'instruction `template` pour déclarer que notre classe `Array` est un type paramétré :

```
template <class T>    // déclare le modèle et le paramètre
class Array           // La classe qui est paramétrée
{
   public :
      Array();
   // Ici, déclaration complète de la classe
};
```

Ce code forme la base de la déclaration d'une classe modèle `Array`. Le mot-clé `template` débute chaque déclaration et définition d'une classe modèle. Il est suivi des paramètres du modèle, qui changeront avec chaque instance, exactement comme pour les fonctions. Dans le modèle de tableau précédent, par exemple, on souhaite que le type des objets stockés dans le tableau puisse être modifié. Une instance pourra stocker un tableau d'entiers, par exemple, alors qu'une autre stockera un tableau d'animaux.

Dans notre exemple, le mot-clé `class` est suivi de l'identificateur `T`. Comme on l'a indiqué plus haut, `class` signifie que ce paramètre est un *type*. L'identificateur `T` est utilisé dans la suite de la définition du modèle pour désigner le type qui aura été passé en paramètre. Cette classe étant maintenant un modèle, une instance pourra par exemple remplacer `T` par `int` et une autre le remplacera par `Chat`. S'il est écrit correctement, le modèle doit pouvoir accepter n'importe quel type de données (ou classe) valide comme valeur de `T`.

On fixe le type pour le modèle lorsque l'on déclare la variable qui en sera une instance. On peut, pour cela, utiliser la syntaxe suivante :

```
nomClasse<type> instance;
```

Ici, `nomClasse` correspond au nom du modèle, `instance` représente le nom de l'instance ou de l'objet créé et `type` est le type de données à utiliser pour l'instance créée.

Voici, par exemple, la déclaration d'une instance `int` et d'une instance `Animal` de la classe tableau paramétrée :

```
Array<int> unTableauInt;
Array<Animal> unTableauAnimal;
```

L'objet `unTableauInt` est de type *tableau d'entiers* et l'objet `unTableauAnimal` est de type *tableau d'animaux*. Vous pouvez maintenant utiliser le type `Array<int>` partout où vous utiliseriez un type en temps normal – comme type du résultat ou d'un paramètre d'une fonction, etc. Le Listing 19.1 présente une déclaration complète d'un modèle de tableau simplifié (attention, ce n'est pas un programme complet, mais plutôt un code consacré à la définition du modèle).

Listing 19.1 : Un modèle de classe *Array*

```
 0:  // Listing 19.1 modèle d'une classe de tableaux
 1:  #include <iostream>
 2:  using namespace std;
 3:  const int TailleDefaut = 10;
 4:
 5:  template <class T>  // déclare le modèle et son paramètre
 6:  class Array         // la classe qui est paramétrée
 7:  {
 8:  public:
 9:     // constructeurs
10:    Array(int taille = TailleDefaut);
11:    Array(const Array &rhs);
12:    ~Array() { delete [] pType; }
13:
14:    // opérateurs
15:    Array& operator=(const Array&);
16:    T& operator[](int indice) { return pType[indice]; }
17:
18:    // méthode d'accès
19:    int GetTaille() { return saTaille; }
20:
21:  private:
22:     T *pType;
23:     int saTaille;
24:  };
```

La déclaration du modèle commence à la ligne 5. Le paramètre, identifié par le mot-clé `class`, est un type et l'identificateur `T` représente le type paramétré. Comme on l'a déjà précisé, vous auriez également pu utiliser le mot `typename` à la place de `class` :

```
5:  template <typename T>  // déclare le modèle et son paramètre
```

Utilisez ce qui vous semble le plus clair, bien qu'il soit préférable d'employer `class` lorsque le type est une classe et `typename` dans les autres cas.

La suite de la déclaration ressemble à n'importe quelle déclaration de classe dans laquelle le type de l'objet serait remplacé par `T`. Par exemple, `operator[]` est déclaré pour renvoyer une référence à un objet de type `T` car, d'habitude, il renvoie une référence à un objet du tableau.

Quand une instance d'un tableau d'entiers sera définie, `T` sera remplacé par un entier ; la méthode `operator[]`, qui est fournie pour ce tableau, renverra donc une référence d'entier, ce qui équivaudra à :

```
int& operator[](int indice) { return pType[indice]; }
```

Lorsqu'une instance d'un tableau d'`Animal` sera déclarée, `operator[]` renverra une référence d'un `Animal` :

```
Animal& operator[](int indice) { return pType[indice]; }
```

Ceci ressemble beaucoup au fonctionnement des macros et, en fait, les modèles ont été créés pour réduire le nombre de macros en C++.

Utilisation du nom

Dans la déclaration de la classe, le mot `Array` peut être utilisé sans autre qualification. Ailleurs dans le programme, cette classe sera désignée par `Array<T>`. Si, par exemple, vous ne codez pas le constructeur dans la déclaration de la classe, la déclaration de cette méthode devra être écrite de la façon suivante :

```
template <class T>
Array<T>::Array(int taille): saTaille = taille
{
  pType = new T[taille];
  for (int i = 0; i < taille; i++)
    pType[i] = 0;
}
```

Comme elle fait partie d'un modèle, la première ligne de cette déclaration est nécessaire pour identifier le type pour la fonction (`class T`). À la deuxième ligne, le nom du modèle est `Array<T>`, et le nom de la fonction est `Array(int taille)`. Vous pouvez également constater que cette méthode attend un entier en paramètre.

La suite de cette fonction ne présente pas de caractéristiques particulières, sauf qu'on emploie le paramètre `T` partout où le type des éléments du tableau doit être utilisé.



Une pratique courante consiste à déclarer normalement la classe et ses méthodes avant de la transformer en modèle. Le développement s'en trouve simplifié et cela permet de se concentrer tout d'abord sur l'objectif de programmation, puis de généraliser la solution avec les modèles.

Vous devez définir les méthodes du modèle dans le fichier où il est déclaré. À la différence des autres classes, où il est possible de placer la déclaration d'une classe et de ses fonctions membres dans un fichier en-tête et leurs définitions dans un fichier `.cpp`, les modèles exigent que ces deux parties se trouvent ensemble soit dans un fichier en-tête, soit dans un fichier `.cpp`. Si vous partagez le modèle avec d'autres parties de votre projet, il est courant de définir les fonctions membres en ligne, dans la déclaration du modèle, ou de les définir juste après la déclaration de la classe, dans le fichier en-tête.

Implémentation du modèle

Après avoir défini un modèle, vous voudrez l'utiliser. L'implémentation complète du tableau de la classe modèle nécessite l'implémentation du constructeur de copie, de `operator=`, etc. Le Listing 19.2 vous présente le code du modèle `Array` ainsi qu'un programme de test simple, qui utilise ce modèle.



Bien que les modèles fassent partie du standard ANSI C++, certains anciens compilateurs ne les supportent pas. Si votre compilateur en fait partie, vous devrez installer sa version la plus récente si vous voulez compiler les exercices de ce chapitre. Cela ne doit pas vous empêcher d'étudier ce chapitre, quitte à y revenir plus tard quand votre compilateur aura été mis à jour.

Listing 19.2 : Implémentation du modèle *Array*

```
0:    // Listing 19.2 : Implémentation du modèle Array
1:    #include <iostream>
2:
3:    const int TailleDefaut= 10;
4:
5:    // déclaration d'une classe simple Animal pour pouvoir
6:    // créer un tableau d'animaux
7:
8:    class Animal
9:    {
10:   public:
11:      Animal(int);
12:      Animal();
13:      ~Animal() {}
14:      int GetPoids() const { return sonPoids; }
15:      void Affiche() const { std::cout << sonPoids; }
16:   private:
17:      int sonPoids;
18:   };
19:
20:   Animal::Animal(int poids):
21:     sonPoids(poids)
22:   {}
23:
24:   Animal::Animal():
25:     sonPoids(0)
26:   {}
27:
28:
29:   template <class T>   // déclare le modèle et son paramètre
30:   class Array          // la classe qui est paramétrée
```

```
31:    {
32:    public:
33:       // constructeurs
34:       Array(int taille = TailleDefaut);
35:       Array(const Array &rhs);
36:       ~Array() { delete [] pType; }
37:
38:       // opérateurs
39:       Array& operator=(const Array&);
40:       T& operator[](int indice) { return pType[indice]; }
41:       const T& operator[](int indice) const
42:       { return pType[indice]; }
43:       // méthode d'accès
44:       int GetTaille() const { return saTaille; }
45:
46:    private:
47:       T *pType;
48:       int saTaille;
49:    };
50:
51:    // Implémentations
52:
53:    // implémentation du Constructeur
54:    template <class T>
55:    Array<T>::Array(int taille ):
56:      saTaille(taille)
57:    {
58:       pType = new T[taille];
59:       // les constructeurs du type que vous créez
60:       // doivent fixer une valeur par défaut
61:    }
62:
63:    // constructeur copie
64:    template <class T>
65:    Array<T>::Array(const Array &rhs)
66:    {
67:       saTaille = rhs.GetTaille();
68:       pType = new T[saTaille];
69:       for (int i = 0; i < saTaille; i++)
70:          pType[i] = rhs[i];
71:    }
72:
73:    // operator=
74:    template <class T>
75:    Array<T>& Array<T>::operator=(const Array &rhs)
76:    {
77:       if (this == &rhs)
78:          return *this;
```

```
 79:       delete [] pType;
 80:       saTaille = rhs.GetTaille();
 81:       pType = new T[saTaille];
 82:       for (int i = 0; i < saTaille; i++)
 83:          pType[i] = rhs[i];
 84:       return *this;
 85:    }
 86:
 87:    // programme de test
 88:    int main()
 89:    {
 90:       Array<int> unTableau;     // un tableau d'entiers
 91:       Array<Animal> unZoo;      // un tableau d'animaux
 92:       Animal *pAnimal;
 93:
 94:       // remplissage des tableaux
 95:       for (int i = 0; i < unTableau.GetTaille(); i++)
 96:       {
 97:         unTableau[i] = i*2;
 98:         pAnimal = new Animal(i*3);
 99:         unZoo[i] = *pAnimal;
100:         delete pAnimal;
101:      }
102:      // Affichage du contenu des tableaux
103:      for (int j = 0; j < unTableau.GetTaille(); j++)
104:      {
105:         std::cout << "unTableau[" << j << "]:\t";
106:         std::cout << unTableau[j] << "\t\t";
107:         std::cout << "unZoo[" << j << "]:\t";
108:         unZoo[j].Affiche();
109:         std::cout << std::endl;
110:      }
111:
112:      return 0;
113:   }
```

Ce programme produit le résultat suivant :

```
unTableau[0]:0unZoo[0]:0
unTableau[1]:2unZoo[1]:3
unTableau[2]:4unZoo[2]:6
unTableau[3]:6unZoo[3]:9
unTableau[4]:8unZoo[4]:12
unTableau[5]:10unZoo[5]:15
unTableau[6]:12unZoo[6]:18
unTableau[7]:14unZoo[7]:21
unTableau[8]:16unZoo[8]:24
unTableau[9]:18unZoo[9]:27
```

Ce programme est assez basique, mais il illustre bien la création et l'utilisation d'un modèle. Ici, on définit et on utilise le modèle `Array` pour créer des instances d'objets `Array` de types `int` et `Animal`. Le tableau d'entiers est rempli d'entiers valant deux fois la valeur de l'indice. L'objet `Array` constitué d'objets `Animal` est appelé `unZoo` ; il est rempli de valeurs égales à trois fois la valeur de l'index.

Les lignes 8 à 26 fournissent une classe `Animal` minimale dont le seul but est d'illustrer la manipulation d'un type défini par l'utilisateur avec un modèle.

L'instruction de ligne 29 débute la déclaration du modèle dont le paramètre est un type désigné par `T`. Cette ligne aurait aussi pu être déclarée avec `typename`.

Les lignes 34 et 35 déclarent les deux constructeurs de la classe `Array` ; le premier prend une taille en paramètre, qui vaut par défaut la valeur de la constante `TailleDefaut`.

Les opérateurs d'affectation et d'indexation sont ensuite déclarés. La seule méthode d'accès est `GetTaille()` (ligne 44), qui renvoie la taille du tableau.

Cette interface est, bien sûr, incomplète. Un programme `Array` sérieux nécessiterait, au minimum, des opérateurs pour supprimer des éléments, étendre le tableau, remplir le tableau, etc. Quand vous utiliserez la classe `Array` de la bibliothèque standard, vous découvrirez que toutes ces fonctionnalités ont été fournies. Vous en saurez plus dans ce chapitre.

Les données `private` de la classe du modèle `Array` sont la taille du tableau et le pointeur du tableau d'objets en mémoire.

À partir de la ligne 53 commence l'implémentation de certaines fonctions membres de la classe modèle. Comme elles sont définies à l'extérieur de la déclaration de la classe, il faut indiquer à nouveau qu'elles font partie du modèle à l'aide de la même instruction que l'on avait placée avant la classe (ligne 54). On précise également que `Array` est une classe modèle en incluant le paramètre de type après le nom de la classe : comme on l'a appelé `T` à la ligne 53, on utilise donc `Array<T>` avec les fonctions membres (ligne 55).

Dans une fonction membre, on peut utiliser le paramètre `T` partout où l'on aurait normalement employé le type d'un élément du tableau. La ligne 58, par exemple, définit le pointeur `pType` pour qu'il pointe vers un nouveau tableau d'éléments de type `T`, c'est-à-dire celui déclaré lors de l'instanciation d'un objet avec cette classe modèle. Lorsqu'un élément d'un type donné est créé, sa construction doit l'initialiser.

Cette procédure se répète avec la déclaration du constructeur de copie aux lignes 64 à 71 et avec la surcharge de l'opérateur d'affectation aux lignes 74 à 85.

La classe modèle `Array` est, en fait, utilisée aux lignes 90 et 91. La ligne 90 instancie un objet `unTableau`, qui utilise le modèle avec des `int`. La ligne 91 instancie `unZoo` pour être un tableau d'éléments de type `Animal`.

Le reste du programme effectue ce qui a été décrit précédemment et est assez simple à comprendre.

Passer des objets modèles instanciés aux fonctions

Si vous voulez transmettre un objet `Array` à une fonction normale, vous devez transmettre une *instance* particulière de ce tableau, pas un modèle. Pour créer une fonction capable de recevoir une instance spécifique de `Array`, vous devez utiliser une déclaration comme celle-ci :

```
void UneFonction(Array<unType>&);
```

où `UneFonction` est le nom de la fonction à laquelle vous passez l'objet `Array` et `unType` est le type des éléments du tableau. Par conséquent, si `UneFonction()` reçoit un tableau d'entiers en paramètre, vous pouvez écrire :

```
void UneFonction(Array<int>&);   // OK
```

et non :

```
void UneFonction(Array<T>&);     // Erreur !
```

car il n'y a aucun moyen de savoir ce que représente `T&`. Vous ne pouvez pas non plus écrire :

```
void UneFonction(Array &);       // Erreur !
```

parce qu'il n'existe pas de classe `Array`, uniquement un modèle et des instances.

Pour créer des fonctions non membres procurant certains avantages des modèles, vous pouvez déclarer une fonction modèle de la même manière que pour la déclaration d'une classe modèle et la définition d'une fonction membre modèle. Il faut d'abord indiquer que la fonction est un modèle, puis utiliser le paramètre de modèle où l'on aurait employé un type ou un nom de classe :

```
template <class T>
void MaFonctionModele(Array<T>&);   // OK
```

La première ligne indique que la fonction `MaFonctionModele()` est une fonction modèle.

Les fonctions modèles peuvent également prendre en paramètres des instances du modèle, en plus de la forme paramétrée, comme ici :

```
template <class T>
void MonAutreFonction(Array<T>&, Array<int>&);   // OK
```

Vous pouvez remarquer que cette fonction reçoit deux tableaux : un tableau paramétré et un tableau d'entiers. Le premier peut contenir tout type d'objet, mais le second devra toujours être un tableau d'entiers. Nous verrons un peu plus loin un exemple de fonction modèle.

Modèles et amis

Vous avez vu les amis au Chapitre 16. Les classes modèles peuvent déclarer trois types d'amis :

- une fonction ou une classe amie qui n'est pas un modèle ;
- une fonction ou une classe amie modèle générale ;
- une fonction ou une classe amie modèle spécifique à un type.

Les sections suivantes présente les deux premières.

Les classes ou fonctions amies non modèles

Vous pouvez déclarer qu'une classe ou une fonction est une amie de votre classe modèle. Chaque instance de la classe traitera cet amie correctement, comme si la déclaration d'amitié avait été faite dans cette instance particulière.

Le Listing 19.3 ajoute `Introduire()`, une fonction amie très simple, à la définition du modèle de la classe `Array`.

`Introduire()` étant une fonction amie, elle peut donc accéder aux données privées de `Array`, mais cette fonction n'étant pas une fonction modèle, on ne peut lui passer que des tableaux d'entiers.

Listing 19.3 : Une fonction amie non modèle

```
0:  // Listing 19.3 - Fonctions amies spécifiques non modèles
1:
2:  #include <iostream>
3:  using namespace std;
4:
5:  const int TailleDefaut= 10;
6:
7:  // Déclaration d'une classe Animal simple pour créer
8:  // ensuite un tableau d'animaux
9:
```

```
10:  class Animal
11:  {
12:  public:
13:     Animal(int);
14:     Animal();
15:     ~Animal() {}
16:     int GetPoids() const { return sonPoids; }
17:     void Affiche() const { cout << sonPoids; }
18:  private:
19:     int sonPoids;
20:  };
21:
22:  Animal::Animal(int poids):
23:     sonPoids(poids)
24:  {}
25:
26:  Animal::Animal():
27:     sonPoids(0)
28:  {}
29:
30:  template <class T>  // déclaration du modèle et du paramètre
31:  class Array         // la classe à paramétrer
32:  {
33:  public:
34:     // constructeurs
35:     Array(int taille = TailleDefaut);
36:     Array(const Array &rhs);
37:     ~Array() { delete [] pType; }
38:
39:     // opérateurs
40:     Array& operator=(const Array&);
41:     T& operator[](int indice) { return pType[indice]; }
42:     const T& operator[](int indice) const
43:     { return pType[indice]; }
44:     // méthode d'accès
45:     int GetTaille() const { return saTaille; }
46:
47:     // fonction amie
48:     friend void Introduire(Array<int>);
49:
50:  private:
51:     T *pType;
52:     int  saTaille;
53:  };
54:
```

```
55:  // fonction amie. Ce n'est pas un modèle, à utiliser
56:  // uniquement avec des tableaux d'entiers!
57:  void Introduire(Array<int> unTableau)
58:  {
59:     cout << endl << "*** Introduire ***" << endl;
60:     for (int i = 0; i < unTableau.saTaille; i++)
61:        cout << "i: " << unTableau.pType[i] << endl;
62:     cout << endl;
63:  }
64:
65:  // Implémentations...
66:
67:  // implémentation du constructeur
68:  template <class T>
69:  Array<T>::Array(int taille):
70:     saTaille(taille)
71:  {
72:     pType = new T[taille];
73:     // les constructeurs du type créé
74:     // doivent définir une valeur par défaut
75:  }
76:
77:  // constructeur de copie
78:  template <class T>
79:  Array<T>::Array(const Array &rhs)
80:  {
81:     saTaille = rhs.GetTaille();
82:     pType = new T[saTaille];
83:     for (int i = 0; i < saTaille; i++)
84:        pType[i] = rhs[i];
85:  }
86:
87:  // operator=
88:  template <class T>
89:  Array<T>& Array<T>::operator=(const Array &rhs)
90:  {
91:     if (this == &rhs)
92:        return *this;
93:     delete [] pType;
94:     saTaille = rhs.GetTaille();
95:     pType = new T[saTaille];
96:     for (int i = 0; i < saTaille; i++)
97:        pType[i] = rhs[i];
98:     return *this;
99:  }
```

```
100:
101:  // programme de test
102:  int main()
103:  {
104:     Array<int> unTableau;     // un tableau d'entiers
105:     Array<Animal> unZoo;      // un tableau d'animaux
106:     Animal *pAnimal;
107:
108:     // remplissage des tableaux
109:     for (int i = 0; i < unTableau.GetTaille(); i++)
110:     {
111:        unTableau [i] = i*2;
112:        pAnimal = new Animal(i*3);
113:        unZoo[i] = *pAnimal;
114:     }
115:
116:     int j;
117:     for (j = 0; j < unTableau.GetTaille(); j++)
118:     {
119:        cout << "unZoo[" << j << "]:\t";
120:        unZoo[j].Affiche();
121:        cout << endl;
122:     }
123:     cout << "Utilisons maintenant la fonction amie ";
124:     cout << "pour trouver les membres de Array<int>";
125:     Introduire(unTableau);
126:
127:     cout << endl << "Fin." << endl;
128:     return 0;
129:  }
```

Ce programme produit le résultat suivant :

```
unZoo[0]:0
unZoo[1]:3
unZoo[2]:6
unZoo[3]:9
unZoo[4]:12
unZoo[5]:15
unZoo[6]:18
unZoo[7]:21
unZoo[8]:24
unZoo[9]:27
Utilisons maintenant la fonction amie pour trouver les membres de Array<int>
*** Introduire ***
i: 0
```

```
i: 2
i: 4
i: 6
i: 8
i: 10
i: 12
i: 14
i: 16
i: 18


Fin.
```

La fonction amie `Introduire()` a été ajoutée à la déclaration du modèle `Array`. Cet ajout à la ligne 48 signifie que chaque instance d'un `Array` de `int` considérera `Introduire()` comme une fonction amie. Celle-ci aura donc accès aux données membres et aux fonctions privées de l'instance du tableau.

La fonction `Introduire()` est définie de la ligne 57 à la ligne 63. `Introduire()` accède directement à `saTaille` à la ligne 60 et à `pType` à la ligne suivante. Cette utilisation triviale des données membres était inutile puisque que la classe `Array` fournit des méthodes d'accès publiques pour ces données, mais cela permet de montrer comment déclarer des fonctions amies avec les modèles.

Les fonctions ou classes amies modèles générales

Il pourrait être utile d'ajouter un opérateur d'affichage à la classe `Array` pour que les valeurs puissent être envoyées à un flux de sortie et traitées comme il convient en fonction de leur type. La première solution consisterait à déclarer un opérateur d'affichage pour chaque type possible de `Array`, mais cela irait à l'encontre du principe des modèles.

Un opérateur d'insertion valable pour tous les types possibles de `Array` constitue une bien meilleure solution :

```
ostream& operator<< (ostream& Array<T>&);
```

Pour ce faire; il faut déclarer `operator<<` comme étant une fonction modèle :

```
template <class T>
ostream& operator<< (ostream&, Array<T>&)
```

`operator<<` étant désormais une fonction modèle, il ne reste plus qu'à lui fournir une implémentation. Le Listing 19.4 développe le modèle `Array` pour inclure cette déclaration et fournir l'implémentation du modèle `operator<<`.

Listing 19.4 : Utilisation d'un opérateur sur *ostream*

```
 0:  //Listing 19.4 utilisation d'un opérateur sur ostream
 1:  #include <iostream>
 2:  using namespace std;
 3:
 4:  const int TailleDefaut= 10;
 5:
 6:  class Animal
 7:  {
 8:  public:
 9:     Animal(int);
10:     Animal();
11:     ~Animal() {}
12:     int GetPoids() const { return sonPoids; }
13:     void Affiche() const { cout << sonPoids; }
14:  private:
15:     int sonPoids;
16:  };
17:
18:  Animal::Animal(int poids):
19:     sonPoids(poids)
20:  {}
21:
22:  Animal::Animal():
23:     sonPoids(0)
24:  {}
25:
26:  template <class T>  // déclaration du modèle et du paramètre
27:  class Array         // la classe qui est paramétrée
28:  {
29:  public:
30:     // constructeurs
31:     Array(int taille = TailleDefaut);
32:     Array(const Array &rhs);
33:     ~Array() { delete [] pType; }
34:
35:     // opérateurs
36:     Array& operator=(const Array&);
37:     T& operator[](int indice) { return pType[indice]; }
38:     const T& operator[](int indice) const
39:        { return pType[indice]; }
40:     // méthode d'accès
41:     int GetTaille() const { return saTaille; }
42:     // template <class T>
43:     friend ostream& operator<<(ostream&, Array<T>&);
44:
```

```
45:  private:
46:     T *pType;
47:     int  saTaille;
48:  };
49:
50:  template <class T>
51:  ostream& operator<<(ostream& sortie, Array<T>& unTableau)
52:  {
53:     for (int i = 0; i < unTableau.saTaille; i++)
54:        sortie << "[" << i << "] " << unTableau[i] << endl;
55:     return sortie;
56:  }
57:
58:  // Implémentations...
59:
60:  // Implémentation du constructeur
61:  template <class T>
62:  Array<T>::Array(int taille):
63:     saTaille(taille)
64:  {
65:     pType = new T[taille];
66:     for (int i = 0; i < taille; i++)
67:        pType[i] = 0;
68:  }
69:
70:  // constructeur de copie
71:  template <class T>
72:  Array<T>::Array(const Array &rhs)
73:  {
74:     saTaille = rhs.GetTaille();
75:     pType = new T[saTaille];
76:     for (int i = 0; i < saTaille; i++)
77:        pType[i] = rhs[i];
78:  }
79:
80:  // operator=
81:  template <class T>
82:  Array<T>& Array<T>::operator=(const Array &rhs)
83:  {
84:     if (this == &rhs)
85:        return *this;
86:     delete [] pType;
87:     saTaille = rhs.GetTaille();
88:     pType = new T[saTaille];
89:     for (int i = 0; i < saTaille; i++)
90:        pType[i] = rhs[i];
91:     return *this;
```

```
 92:  }
 93:
 94:  int main()
 95:  {
 96:     bool Stop = false;       // indicateur de boucle
 97:     int indice, valeur;
 98:     Array<int> leTableau ;
 99:
100:     while (Stop == false)
101:     {
102:        cout << "Entrez un indice (0-9) ";
103:        cout << "et une valeur. (-1 pour arrêter) : ";
104:        cin >> indice >> valeur;
105:
106:        if (indice < 0)
107:           break;
108:
109:        if (indice > 9)
110:        {
111:           cout << "Entrez un indice entre 0 et 9.\n";
112:           continue;
113:        }
114:
115:        leTableau[indice] = valeur;
116:     }
117:
118:     cout << "\nVoici le tableau complet:\n";
119:     cout << leTableau << endl;
120:     return 0;
121:  }
```



Si vous utilisez un compilateur Windows, décommentez la ligne 42. Selon la norme de C++, cette ligne ne devrait pas être nécessaire, mais elle l'est pour le compilateur Microsoft C++.

Ce programme produit le résultat suivant :

```
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 1 10
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 2 20
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 3 30
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 4 40
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 5 50
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 6 60
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 7 70
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 8 80
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 9 90
```

```
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 10 10
Entrez un indice entre 0 et 9.
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : -1 -1

Voici le tableau complet:
[0] 0
[1] 10
[2] 20
[3] 30
[4] 40
[5] 50
[6] 60
[7] 70
[8] 80
[9] 90
```

La ligne 43 déclare la fonction modèle `operator<<()` comme amie du modèle `Array`. `operator<<()` étant implémentée comme une fonction modèle, chaque instance de son type de tableau paramétré possèdera automatiquement un `operator<<()`.

L'implémentation de cet opérateur commence à la ligne 50. La boucle simple des lignes 53 et 54 utilise tour à tour aux éléments du tableau. Cette technique ne peut évidemment fonctionner que si `operator<<()` est défini pour tous les types d'objet stockés dans le tableau.

Vous remarquerez que ce listing exige également que `operator[]` ait été surchargé. Comme vous pouvez le constater à la ligne 37, c'est ce que nous avons fait en utilisant également le paramètre du type..

Utilisation des des modèles

Les modèles sont traités comme n'importe quel autre type. Vous pouvez les transmettre en paramètres, par référence ou par valeur et vous pouvez les renvoyer dans les résultats des fonctions, également par référence ou par valeur. Le Listing 19.5 illustre la transmission d'objets modèles.

Listing 19.5 : Echange d'objets modèles avec les fonctions

```
0:  //Listing 19.5 Échange d'objets modèles avec les fonctions
1:  #include <iostream>
2:  using namespace std;
3:
4:  const int TailleDefaut= 10;
5:
```

```
 6: // classe simple à ajouter aux tableaux
 7: class Animal
 8: {
 9: public:
10:    // constructeurs
11:    Animal(int);
12:    Animal();
13:    ~Animal();
14:
15:    // méthodes d'accès
16:    int GetPoids() const { return sonPoids; }
17:    void SetPoids(int poids) { sonPoids = poids; }
18:
19:    // opérateurs amis
20:    friend ostream& operator<< (ostream&, const Animal&);
21:
22: private:
23:    int sonPoids;
24: };
25:
26: // opérateur d'insertion pour afficher les animaux
27: ostream& operator<<
28: (ostream& unFlux, const Animal& unAnimal)
29: {
30:    unFlux << unAnimal.GetPoids();
31:    return unFlux;
32: }
33:
34: Animal::Animal(int poids):
35:   sonPoids(poids)
36: {
37:    // cout << "Animal(int)" << endl;
38: }
39:
40: Animal::Animal():
41:   sonPoids(0)
42: {
43:    // cout << "Animal()" << endl;
44: }
45:
46: Animal::~Animal()
47: {
48:    // cout << "Destruction d'un animal..." << endl;
49: }
50:
```

```
51:  template <class T>  // déclaration du modèle et du paramètre
52:  class Array         // la classe qui est paramétrée
53:  {
54:  public:
55:     Array(int taille = TailleDefaut);
56:     Array(const Array &rhs);
57:     ~Array() { delete [] pType; }
58:
59:     Array& operator=(const Array&);
60:     T& operator[](int indice) { return pType[indice]; }
61:     const T& operator[](int indice) const
62:        { return pType[indice]; }
63:     int GetTaille() const { return saTaille; }
64:     // fonction amie
65:     // template <class T>
66:     friend ostream& operator<< (ostream&, const Array<T>&);
67:
68:  private:
69:     T *pType;
70:     int  saTaille;
71:  };
72:
73:  template <class T>
74:  ostream& operator<< (ostream& out, const Array<T>& unTableau)
75:  {
76:     for (int i = 0; i < unTableau.saTaille; i++)
77:        out << "[" << i << "] " << unTableau[i] << endl;
78:     return out;
79:  }
80:
81:  // Partie des implémentations...
82:
83:  // Implémentation du constructeur
84:  template <class T>
85:  Array<T>::Array(int taille):
86:    saTaille(taille)
87:  {
88:     pType = new T[taille];
89:     for (int i = 0; i < taille; i++)
90:        pType[i] = 0;
91:  }
92:
93:  // constructeur de copie
94:  template <class T>
95:  Array<T>::Array(const Array &rhs)
96:  {
97:     saTaille = rhs.GetTaille();
```

```
 98:    pType = new T[saTaille];
 99:    for (int i = 0; i < saTaille; i++)
100:        pType[i] = rhs[i];
101:  }
102:
103:  void RemplirEntiers(Array<int>& unTableau);
104:  void RemplirAnimaux(Array<Animal>& unTableau);
105:
106:  int main()
107:  {
108:     Array<int> tabInt;
109:     Array<Animal> tabAnimal;
110:     RemplirEntiers(tabInt);
111:     RemplirAnimaux(tabAnimal);
112:     cout << "tabInt...\n" << tabInt;
113:     cout << "\ntabAnimal...\n" << tabAnimal << endl;
114:     return 0;
115:  }
116:
117:  void RemplirEntiers(Array<int>& unTableau)
118:  {
119:     bool Stop = false;
120:     int indice, valeur;
121:     while (Stop == false)
122:     {
123:        cout << "Entrez un indice (0-9) ";
124:        cout << "et une valeur. (-1 pour arrêter) : " ;
125:        cin >> indice >> valeur;
126:        if (indice < 0)
127:           break;
128:        if (indice > 9)
129:        {
130:           cout << "Entrez un indice entre 0 et 9.\n";
131:           continue;
132:        }
133:        unTableau[indice] = valeur;
134:     }
135:  }
136:
137:
138:  void RemplirAnimaux(Array<Animal>& unTableau)
139:  {
140:     Animal * pAnimal;
141:     for (int i = 0; i < unTableau.GetTaille(); i++)
142:     {
```

```
143:         pAnimal = new Animal;
144:         pAnimal->SetPoids(i*100);
145:         unTableau[i] = *pAnimal;
146:         delete pAnimal;  // copie placée dans le tableau
147:      }
148:  }
```



Si vous utilisez un compilateur Windows, décommentez la ligne 65. Selon la norme C++, cette ligne ne devrait pas être nécessaire, mais elle l'est pour le compilateur Microsoft C++.

Ce programme produit le résultat suivant :

```
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 1 10
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 2 20
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 3 30
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 4 40
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 5 50
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 6 60
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 7 70
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 8 80
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 9 90
Entrez un indice (0-9) et une valeur. (-1 pour arrêter) : 10 10
Entrez un indice entre 0 et 9.
Entrez un indice (0-9) et une valeur. (-1 pour arrêter): -1 -1

tabInt...
 [0] 0
 [1] 10
 [2] 20
 [3] 30
 [4] 40
 [5] 50
 [6] 60
 [7] 70
 [8] 80
 [9] 90

tabAnimal...
 [0] 0
 [1] 100
 [2] 200
 [3] 300
```

```
[4] 400
[5] 500
[6] 600
[7] 700
[8] 800
[9] 900
```

La classe `Animal` est déclarée des lignes 7 à 24. Bien que cette classe soit très simplifiée, elle contient son propre opérateur d'insertion (<<) pour l'affichage des animaux. Comme vous pouvez le constater, cet affichage se borne à afficher le poids de l'animal.

Vous remarquerez que `Animal` a un constructeur par défaut. Ce constructeur est nécessaire, car c'est lui qui crée l'objet lorsqu'on l'ajoute à un tableau, ce qui pose quelques problèmes, comme nous le verrons bientôt.

La fonction `RemplirEntiers()` est déclarée à la ligne 103. Son prototype indique que cette fonction prend en paramètre un `Array` d'entiers. Ce n'est pas une fonction modèle, car elle ne connaît qu'un type de `Array` : un tableau d'entiers. De façon analogue, la fonction `RemplirAnimaux()` de la ligne 104 est déclarée pour les `Array` d'animaux.

Les implémentations de ces deux fonctions sont différentes parce que remplir un tableau d'entiers ne peut pas se faire de la même façon que le remplissage d'un tableau d'animaux.

Les fonctions spécialisées

Si vous décommentez les instructions d'affichage des lignes 35 et 41 du Listing 19.4, vous constaterez qu'il y a des constructions et des destructions d'animaux que vous n'aviez peut-être pas prévues.

Lorsqu'un objet est ajouté à un tableau, le constructeur par défaut de cet objet est appelé. Le constructeur de `Array`, cependant, continue à attribuer la valeur 0 à chaque élément du tableau, comme le montrent les lignes 89 et 90 du Listing 19.5.

Lorsque l'on écrit `unAnimal = (Animal) 0;`, on appelle en fait la méthode l'`operator=` par défaut pour `Animal`. Cela crée donc un objet temporaire `Animal` en utilisant le constructeur qui reçoit en paramètre la valeur zéro. Cet objet temporaire est utilisé à droite de l'opérateur d'affectation (=) puis détruit.

Cela représente une perte de temps, car l'objet `Animal` était déjà correctement initialisé, mais vous ne pouvez malheureusement pas supprimer cette ligne car les entiers ne sont pas automatiquement initialisés à 0. La solution consiste à demander au modèle de ne pas utiliser ce constructeur pour `Animal`, mais un constructeur `Animal` particulier.

Vous pouvez fournir une implémentation explicite pour la classe `Animal`, comme le montre le Listing 19.6. Ce type de spécification est appelée *spécialisation* du modèle.

Listing 19.6 : Implémentations de modèles spécialisées

```
0:     #include <iostream>
1:     using namespace std;
2:
3:     const int TailleDefaut= 3;
4:
5:     // classe très simple pour ajouter aux tableaux
6:       class Animal
7:       {
8:       public:
9:         // constructeurs
10:        Animal(int);
11:        Animal();
12:        ~Animal();
13:
14:        // méthodes d'accès
15:        int GetPoids() const { return sonPoids; }
16:        void GetPoids(int poids) { sonPoids = poids; }
17:
18:        // opérateur ami
19:        friend ostream& operator<< (ostream&, const Animal&);
20:
21:     private:
22:        int sonPoids;
23:     };
24:
25:      // opérateur d'insertion pour afficher les animaux
26:     ostream& operator<<
27:         (ostream& unFlux, const Animal& unAnimal)
28:     {
29:       unFlux << unAnimal.GetPoids();
30:       return unFlux;
31:     }
32:
33:     Animal::Animal(int poids):
34:       sonPoids(poids)
35:     {
36:        cout << "Animal(int) ";
37:     }
38:
39:     Animal::Animal():
40:       sonPoids(0)
41:     {
42:        cout << "Animal() ";
43:     }
44:
```

```
45:      Animal::~Animal()
46:      {
47:        cout << "Destruction d'un animal...";
48:      }
49:
50:    template <class T>  // déclare le modèle et le paramètre
51:    class Array         // la classe qui est paramétrée
52:    {
53:    public:
54:       Array(int taille = ::TailleDefaut);
55:       Array(const Array &rhs);
56:       ~Array() { delete [] pType; }
57:
58:       // opérateurs
59:       Array& operator=(const Array&);
60:       T& operator[](int indice) { return pType[indice]; }
61:       const T& operator[](int indice) const
62:          { return pType[indice]; }
63:
64:       // méthode d'accès
65:       int GetTaille() const { return saTaille; }
66:       // fonction amie
67:       // template <class T>
68:       friend ostream& operator<< (ostream&, const Array<T>&);
69:
70:    private:
71:       T *pType;
72:       int  saTaille;
73:    };
74:
75:  template <class T>
76:  Array<T>::Array(int taille):
77:    saTaille(taille)
78:  {
79:     pType = new T[taille];
80:     for (int i = 0; i < taille; i++)
81:        pType[i] = (T)0;
82:  }
83:
84:  template <class T>
85:  Array<T>& Array<T>::operator=(const Array &rhs)
86:  {
87:     if (this == &rhs)
88:        return *this;
89:     delete [] pType;
90:     saTaille = rhs.GetTaille();
91:     pType = new T[saTaille];
```

```
 92:     for (int i = 0; i < saTaille; i++)
 93:        pType[i] = rhs[i];
 94:     return *this;
 95:  }
 96:
 97:  template <class T>
 98:  Array<T>::Array(const Array &rhs)
 99:  {
100:     saTaille = rhs.GetTaille();
101:     pType = new T[saTaille];
102:     for (int i = 0; i < saTaille; i++)
103:        pType[i] = rhs[i];
104:  }
105:
106:
107:  template <class T>
108:  ostream& operator<<(ostream& out, const Array<T>& unTableau)
109:  {
110:     for (int i = 0; i < unTableau.GetTaille(); i++)
111:        out << "[" << i << "] " << unTableau[i] << endl;
112:     return out;
113:  }
114:
115:
116:  Array<Animal>::Array(int tailleTabAnimaux):
117:    saTaille(tailleTabAnimaux)
118:  {
119:     pType = new Animal[tailleTabAnimaux];
120:  }
121:
122:
123:  void RemplirEntiers(Array<int>& unTableau);
124:  void RemplirAnimaux(Array<Animal>& unTableau);
125:
126:  int main()
127:  {
128:     Array<int> tabInt;
129:     Array<Animal> tabAnimal;
130:     RemplirEntiers(tabInt);
131:     RemplirAnimaux(tabAnimal);
132:     cout << "tabInt...\n" << tabInt;
133:     cout << "\ntabAnimal...\n" << tabAnimal << endl;
134:     return 0;
135:  }
136:
137:  void RemplirEntiers(Array<int>& unTableau)
138:  {
```

```
139:     bool Stop = false;
140:     int indice, valeur;
141:     while (Stop == false)
142:     {
143:        cout << "Entrez un indice (0-2) et une valeur. ";
144:        cout << "(-1 pour arrêter): " ;
145:        cin >> indice >> valeur;
146:        if (indice < 0)
147:           break;
148:        if (indice > 2)
149:        {
150:           cout << "Entrer un indice entre 0 et 2.\n";
151:           continue;
152:        }
153:        unTableau[indice] = valeur;
154:     }
155:  }
156:
157:
158:  void RemplirAnimaux(Array<Animal>& unTableau)
159:  {
160:     Animal * pAnimal;
161:     for (int i = 0; i < unTableau.GetTaille(); i++)
162:     {
163:        pAnimal = new Animal(i*10);
164:        unTableau[i] = *pAnimal;
165:        delete pAnimal;
166:     }
167:  }
```



Si vous utilisez un compilateur Windows, décommentez la ligne 67. Selon la norme C++, cette ligne ne devrait pas être nécessaire, mais elle l'est pour le compilateur Microsoft C++.

Les numéros de lignes ont été ajoutés à la sortie de ce programme afin d'en faciliter l'analyse. Vous ne les verrez pas apparaître sur votre écran.

Ce programme produit le résultat suivant.

Première exécution :

```
1:  Animal() Animal() Animal() Entrez un indice (0-2) et une valeur.
    (-1 pour arrêter): 0 0
2:  Entrez un indice (0-2) et une valeur. (-1 pour arrêter): 0 1
3:  Entrez un indice (0-2) et une valeur. (-1 pour arrêter): 1 2
4:  Entrez un indice (0-2) et une valeur. (-1 pour arrêter): 2 3
```

```
5:  Entrez un indice (0-2) et une valeur. (-1 pour arrêter): -1 -1
6:  Animal(int) Destruction d'un animal...Animal(int) Destruction d'un
animal...Animal(int) Destruction d'un animal...tabInt...
7: [0] 0
8: [1] 1
9: [2] 2
10:
11: tabAnimal...
12: [0] 0
13: [1] 10
14: [2] 20
15:
16: Destruction d'un animal...Destruction d'un animal...Destruction d'un
animal...
```

Seconde exécution :

```
 1:  Animal(int)  Destruction d'un animal...
 2:  Animal(int)  Destruction d'un animal...
 3:  Animal(int)  Destruction d'un animal...
 4:  Entrez un indice (0-9) et une valeur. (-1 pour arrêter): 0 0
 5:  Entrez un indice (0-9) et une valeur. (-1 pour arrêter): 1 1
 6:  Entrez un indice (0-9) et une valeur. (-1 pour arrêter): 2 2
 7:  Entrez un indice (0-9) et une valeur. (-1 pour arrêter): 3 3
 8:  Animal(int)
 9:  Destruction d'un animal...
10: Animal(int)
11: Destruction d'un animal...
12: Animal(int)
13: Destruction d'un animal...
14: tabInt...
15: [0] 0
16: [1] 1
17: [2] 2
18:
19: tabAnimal...
20: [0] 0
21: [1] 10
22: [2] 20
23:
24: Destruction d'un animal...
25: Destruction d'un animal...
26: Destruction d'un animal...
```

Le Listing 19.6 reproduit les deux classes dans leur intégralité pour illustrer la création et la destruction des objets `Animal` temporaires. La valeur de `TailleDefaut` a été réduite de 10 à 3 pour simplifier la sortie.

Les constructeurs et destructeurs de `Animal`, aux lignes 33 à 48, affichent chacun un message qui signale leur appel.

Un constructeur `Array` est déclaré lignes 75 à 82 en tant que modèle. Les lignes 116 à 120 montrent comment créer un constructeur spécialisé pour un `Array` d'animaux. Vous remarquerez que, dans ce constructeur spécial, le constructeur par défaut peut définir la valeur initiale de chaque `Animal` et qu'il n'y a aucune affectation explicite.

La première exécution de ce programme donne la première série de messages. La première ligne montre les trois constructeurs par défaut qui sont appelés pour créer le tableau. L'utilisateur entre ensuite quatre nombres qui sont stockés dans le tableau d'entiers.

L'exécution se poursuit avec `RemplirAnimaux()`. Un objet `Animal` temporaire est alors créé à la ligne 163 et sa valeur sert à modifier l'objet `Animal` du tableau à la ligne suivante. Cet `Animal` temporaire est détruit à la ligne 165. Les messages affichés par la ligne 6 indiquent que ce processus se répète pour chaque élément du tableau.

Les tableaux sont détruits à la fin du programme. Lorsque leurs destructeurs sont appelés, tous les objets qu'ils contiennent sont également détruits, comme le montrent les messages de la ligne 16.

Pour la seconde série de messages, on a mis en commentaire l'implémentation spéciale du constructeur du tableau d'animaux, aux lignes 116 à 120 du programme. Lorsque le programme est de nouveau exécuté, c'est donc le constructeur modèle (lignes 75 à 82 du programme) qui est appelé quand le tableau `Animal` est construit. Cela provoque l'appel d'objets `Animal` temporaires pour chaque élément du tableau aux lignes 80 et 81 du programme (messages des lignes 1 à 3 de la seconde série de résultats).

La suite du déroulement du programme est identique à celle de la première exécution.

Membres statiques et modèles

Un modèle peut déclarer des données membres statiques. Un ensemble unique de données statiques sera alors créé pour chaque type de classe créée à partir du modèle. Ceci signifie que si vous ajoutez un membre statique à la classe `Array` (un compteur du nombre de tableaux créés, par exemple), ce membre existera pour chaque type : un membre pour tous les tableaux d'animaux et un autre pour tous les tableaux d'entiers. Le Listing 19.7 ajoute un membre statique et une fonction statique à la classe `Array`.

Listing 19.7 : Utilisation de fonctions membres et de données membres statiques avec les modèles

```
 0:  #include <iostream>
 1:  using namespace std;
 2:
 3:  const int TailleDefaut= 3;
 4:
 5:  // Une classe simple à ajouter aux tableaux
 6:  class Animal
 7:  {
 8:  public:
 9:     // constructeurs
10:     Animal(int);
11:     Animal();
12:     ~Animal();
13:
14:     // Méthodes d'accès
15:     int GetPoids() const { return sonPoids; }
16:     void SetPoids(int poids) { sonPoids = poids; }
17:
18:     // Opérateur ami
19:     friend ostream& operator<< (ostream&, const Animal&);
20:
21:  private:
22:     int sonPoids;
23:  };
24:
25:  // opérateur d'insertion pour afficher les animaux
26:  ostream& operator<<
27:     (ostream& unFlux, const Animal& unAnimal)
28:  {
29:     unFlux << unAnimal.GetPoids();
30:     return unFlux;
31:  }
32:
33:  Animal::Animal(int poids):
34:     sonPoids(poids)
35:  {
36:     //cout << "Animal(int) ";
37:  }
38:
39:  Animal::Animal():
40:     sonPoids(0)
41:  {
```

```
42:     //cout << "Animal() ";
43:  }
44:
45:  Animal::~Animal()
46:  {
47:     //cout << "Un animal détruit...";
48:  }
49:
50:     template <class T>  // déclare le modèle et le paramètre
51:     class Array         // la classe qui est paramétrée
52:     {
53:     public:
54:        // constructeurs
55:        Array(int taille = TailleDefaut);
56:        Array(const Array &rhs);
57:       ~Array() { delete [] pType; sonNbreArray--; }
58:
59:       // opérateurs
60:       Array& operator=(const Array&);
61:       T& operator[](int indice) { return pType[indice]; }
62:     const T& operator[](int indice) const
63:     { return pType[indice]; }
64:       // méthodes d'accès
65:       int GetTaille() const { return saTaille; }
66:       static int GetNbreArray() { return sonNbreArray; }
67:
68:       // fonction amie
69:      friend ostream& operator<< (ostream&, const Array<T>&);
70:
71:    private:
72:       T *pType;
73:       int  saTaille;
74:       static int sonNbreArray;
75:    };
76:
77:    template <class T>
78:       int Array<T>::sonNbreArray = 0;
79:
80:    template <class T>
81:    Array<T>::Array(int taille):
82:     saTaille(taille)
83:  {
84:     pType = new T[taille];
85:     for (int i = 0; i < taille; i++)
```

```
 86:        pType[i] = (T)0;
 87:     sonNbreArray++;
 88:  }
 89:
 90:  template <class T>
 91:  Array<T>& Array<T>::operator=(const Array &rhs)
 92:  {
 93:     if (this == &rhs)
 4:        return *this;
 95:     delete [] pType;
 96:     saTaille = rhs.GetTaille();
 97:     pType = new T[saTaille];
 98:     for (int i = 0; i < saTaille; i++)
 99:        pType[i] = rhs[i];
100:  }
101:
102:  template <class T>
103:  Array<T>::Array(const Array &rhs)
104:  {
105:     saTaille = rhs.GetTaille();
106:     pType = new T[saTaille];
107:     for (int i = 0; i < saTaille; i++)
108:        pType[i] = rhs[i];
109:     sonNbreArray++;
110:  }
111:
112:  template <class T>
113:  ostream& operator<<(ostream& out, const Array<T>& unTableau)
114:  {
115:     for (int i = 0; i < unTableau.GetTaille(); i++)
116:        out << "[" << i << "] " << unTableau[i] << endl;
117:     return out;
118:  }
119:
120:  int main()
121:  {
122:       cout << Array<int>::GetNbreArray()
122a:            << " tableaux d'entiers\n";
123:       cout << Array<Animal>::GetNbreArray();
124:        cout << " tableaux d'animaux" << endl << endl;
125:       Array<int> tabInt;
126:       Array<Animal> tabAnimal;
127:
128:       cout << tabInt.GetNbreArray()
128a:            << " tableaux d'entiers\n";
```

```
129:       cout << tabAnimal.GetNbreArray();
130:       cout << " tableaux d'animaux" << endl << endl;
131:
132:       Array<int> *pTabInt = new Array<int>;
133:
134:       cout << Array<int>::GetNbreArray()
134a:           << " tableaux d'entiers\n";
135:       cout << Array<Animal>::GetNbreArray();
136:       cout << " tableaux d'animaux" << endl << endl;
137:
138:       delete pTabInt;
139:
140:      cout << Array<int>::GetNbreArray()
140a:          << " tableaux d'entiers\n";
141:      cout << Array<Animal>::GetNbreArray();
142:      cout << " tableaux d'animaux" << endl << endl;
143:     return 0;
144:   }
```

Ce programme produit le résultat suivant :

```
0 tableau d'entiers
0 tableau d'animaux

1 tableau d'entiers
1 tableau d'animaux

2 tableaux d'entiers
1 tableau d'animaux

1 tableau d'entiers
1 tableau d'animaux
```

La ligne 74 ajoute la variable statique `sonNbreArray` à la classe `Array`. Cette donnée étant privée, la méthode d'accès statique `GetNbreArray()` a été ajoutée à la ligne 66.

L'initialisation de la donnée statique est réalisée aux lignes 77 et 78, avec une qualification complète qui mentionne le modèle. Les constructeurs et le destructeur de `Array` ont également été modifiés pour mémoriser le nombre de tableaux existants à un instant donné.

L'accès à ces membres statiques est identique à celui des membres statiques de n'importe quelle classe : il peut être réalisé avec un objet existant, comme le montrent les lignes 134 et 135 ou en utilisant la spécification complète de la classe, comme le montrent les

lignes 128 et 129. Vous remarquerez que vous devez utiliser un type de tableau spécifique pour accéder aux données statiques. Il n'existe qu'une seule variable pour chaque type.

Faire	*Ne pas faire*
● Utiliser des modèles à chaque fois que vous avez un concept pouvant fonctionner sur des objets de différentes classes ou sur différents types de données primitives. ● Utiliser les paramètres des fonctions modèles pour que leurs instances aient le type correct. ● Utiliser des membres statiques avec les modèles si vous en avez besoin. ● Spécialiser le comportement du modèle en redéfinissant les fonctions modèles par type.	● Ne pas continuer à vous intéresser aux modèles. Ce chapitre n'a présenté que quelques-unes des applications des modèles. Une présentation complète sort du cadre de ce livre. ● Vous inquiéter si vous n'avez pas tout compris les techniques de création des modèles. Pour l'instant, il est plus important de savoir les utiliser. Comme vous le verrez dans la prochaine section, la STL contient un grand nombre de modèles prêts à l'emploi.

La bibliothèque des modèles standard

Comme nous l'avons déjà dit, il est inutile de réinventer la roue. C'est pourquoi les programmeurs C++ ont décidé d'utiliser la STL (*Standard Template Library*), qui rassemble un grand nombre d'outils puissants, permettant de traiter les tâches de programmation classique.

Les principaux compilateurs disponibles aujourd'hui incluent cette bibliothèque. Elle contient des modèles de conteneurs, comme les vecteurs, les listes, les files, et les piles. La STL contient également un certain nombre d'algorithmes classiques, comme les tris et les recherches.

Cette bibliothèque a été testée et déboguée, elle offre des performances élevées et elle est disponible gratuitement mais, surtout, elle est réutilisable : il suffit de comprendre comment utiliser un conteneur de la STL pour pouvoir s'en servir dans tous ses programmes sans devoir le réinventer.

Les conteneurs

Un *conteneur* est un objet qui en contient d'autres. La bibliothèque standard de C++ fournit une série de classes conteneur à partir desquelles les développeurs C++ peuvent traiter les tâches de programmation courantes.

Les deux types de classes conteneur de la STL sont les conteneurs séquentiels et les conteneurs associatifs. Les *séquences* sont conçues pour offrir un accès séquentiel et direct à

leurs membres, ou *éléments*. Les *conteneurs associatifs* sont optimisés pour permettre un accès à leurs éléments à partir de valeurs clés. Toutes les classes conteneurs de cette bibliothèque sont définies dans l'espace de nom `std`.

Les conteneurs séquentiels

Avec les conteneurs séquence de la STL, vous disposez d'un accès séquentiel efficace à une liste d'objets. La bibliothèque C++ standard fournit cinq conteneurs séquence : `vector`, `list`, `stack`, `deque` et `queue`.

Le conteneur *vector*

On utilise souvent des tableaux pour stocker et accéder à un certain nombre d'éléments. Les éléments d'un tableau sont tous du même type et sont accessibles à partir d'un indice. La classe conteneur `vector` fournie par la STL se comporte comme un tableau, mais offre davantage de puissance et de sécurité qu'un tableau classique de C++.

Un `vector` est un conteneur optimisé pour offrir un accès rapide à ses éléments *via* un indice. Cette classe est définie dans le fichier en-tête `<vector>` de l'espace de nom `std` (les espaces de noms sont détaillés au Chapitre 18).

La taille d'un `vector` évolue automatiquement en fonction des besoins. Supposons que vous ayez créé un `vector` de 10 éléments. Lorsque vous aurez stocké 10 objets, ce `vector` sera plein ; si vous ajoutez alors un autre objet, le vector augmentera automatiquement sa capacité pour recevoir le onzième objet. Voici comment la classe `vector` est définie :

```
template <class T, class Allocator = allocator<T>> class vector
{
    // membres de la classe
};
```

Le premier paramètre (`class T`) représente le type des éléments du `vector`. Le second (`class Allocator`) est une classe allocateur. Les *allocateurs* sont des gestionnaires de mémoire chargés d'allouer et libérer la mémoire requise pour les éléments des conteneurs. Le concept d'allocateur et l'implémentation de ces gestionnaires sont des sujets très pointus qui sortent du cadre de cet ouvrage.

Par défaut, les éléments sont créés avec l'opérateur `new()` et sont libérés avec l'opérateur `delete()`. Ceci signifie que c'est le constructeur par défaut de la classe `T` qui est appelé pour créer un nouvel élément. C'est donc encore une autre bonne raison de définir explicitement un constructeur par défaut pour ses propres classes. Si vous ne le faites pas, vous ne pourrez pas utiliser le conteneur `vector` standard pour stocker des instances de votre classe.

Voici comment définir des `vector` pour y stocker des entiers et des nombres à virgule flottante :

```
vector<int> vInts;        // vector d'entiers
vector<float> vFloats;    // vector de flottants
```

Quand vous créez un `vector`, vous avez généralement une idée du nombre d'éléments qu'il devra contenir. Supposons, par exemple, que le nombre maximal d'étudiants par classe dans votre école soit de 50. Si vous créez un `vector` pour y stocker les étudiants d'une classe, celui-ci devra être capable de recevoir jusqu'à 50 éléments. La classe `vector` standard fournit un constructeur auquel vous pouvez transmettre le nombre d'éléments en paramètre. Voici comment vous pouvez définir ce `vector` de 50 étudiants :

```
vector<Etudiant> ClasseMaths(50);
```

Le compilateur allouera suffisamment de mémoire pour enregistrer 50 étudiants. Chacun d'eux sera créé avec le constructeur par défaut `Etudiant::Etudiant()`.

Vous pouvez connaître le nombre d'éléments d'un `vector` avec sa fonction membre `size()`. Pour le vecteur d'`Etudiant` `ClasseMaths` qui vient d'être défini, `Classe.size()` renverra 50.

Une autre fonction membre, `capacity()`, permet de savoir combien d'éléments un `vector` peut recevoir avant que sa capacité n'ait besoin d'être augmentée. Nous développerons ce sujet un peu plus loin.

On dit qu'un `vector` est vide s'il ne contient aucun élément, c'est-à-dire si sa taille est nulle. La classe `vector` fournit la fonction membre `empty()` pour tester facilement si le conteneur est vide. Elle renvoie `true` si le vecteur est vide.

Pour affecter un objet `Etudiant Jean` à la `Classe`, vous pouvez utiliser l'opérateur d'indexation :

```
ClasseMaths[5] = Jean;
```

Les indices commencent à zéro. Vous avez certainement remarqué que nous avons utilisé ici l'opérateur d'affectation surchargé de la classe `Etudiant` pour affecter `Jean` au 6e élément de `Classe`. De la même façon, pour trouver l'âge de `Jean`, vous pouvez accéder à son enregistrement en écrivant, par exemple:

```
ClasseMaths[5].GetAge();
```

Comme nous l'avons déjà mentionné, les `vector` sont capables d'augmenter automatiquement leur capacité quand vous ajoutez plus d'éléments qu'ils ne sont capables d'en recevoir. Supposons qu'une des classes de votre école devienne subitement très populaire et que le nombre d'étudiants excède les 50. Quand le 51e étudiant, `Michel`, est ajouté dans

ClasseMaths, le vecteur peut augmenter sa taille pour recevoir cet élément supplémentaire.

Il existe plusieurs méthodes pour ajouter un élément dans un vector. L'une d'entre elles consiste à utiliser la fonction push_back() :

```
ClasseMaths.push_back(Michel);
```

Cette fonction membre ajoute le nouvel objet Etudiant Michel à la fin du vector ClasseMaths. Ce dernier contient maintenant 51 éléments et Michel est l'élément ClasseMaths[50].

Pour que cette fonction s'exécute correctement, la classe Etudiant doit définir un constructeur de copie. Sinon, push_back() sera incapable d'effectuer une copie de l'objet Michel.

La STL ne précise pas le nombre maximal d'éléments qu'un vector peut recevoir ; les éditeurs de compilateur sont les mieux placés pour prendre cette décision. La classe vector fournit une fonction membre qui vous donne ce précieux renseignement pour votre compilateur : max_size().

Le Listing 19.8 met en œuvre les membres de la classe vector que nous venons de présenter. On utilise ici la classe string standard pour simplifier la gestion des chaînes. Pour plus de détails sur la classe string, consultez la documentation de votre compilateur.

Listing 19.8 : Création d'un vecteur et accès à ses éléments

```
 0:  #include <iostream>
 1:  #include <string>
 2:  #include <vector>
 3:  using namespace std;
 4:
 5:  class Etudiant
 6:  {
 7:  public:
 8:     Etudiant();
 9:     Etudiant(const string& nom, const int age);
10:     Etudiant(const Etudiant& rhs);
11:     ~Etudiant();
12:
13:     void   SetNom(const string& nom);
14:     string GetNom()   const;
15:     void   SetAge(const int age);
16:     int    GetAge()   const;
17:
18:     Etudiant& operator=(const Etudiant& rhs);
19:
```

```
20:  private:
21:     string sonNom;
22:     int sonAge;
23:  };
24:
25:  Etudiant::Etudiant() :
26:    sonNom("Nouvel étudiant"), sonAge(16)
27:  {}
28:
29:  Etudiant::Etudiant(const string& nom, const int age) :
30:    sonNom(nom), sonAge(age)
31:  {}
32:
33:  Etudiant::Etudiant(const Etudiant& rhs) :
34:    sonNom(rhs.GetNom()), sonAge(rhs.GetAge())
35:  {}
36:
37:  Etudiant::~Etudiant()
38:  {}
39:
40:  void Etudiant::SetNom(const string& nom)
41:  {
42:     sonNom = nom;
43:  }
44:
45:  string Etudiant::GetNom() const
46:  {
47:     return sonNom;
48:  }
49:
50:  void Etudiant::SetAge(const int age)
51:  {
52:     sonAge = age;
53:  }
54:
55:  int Etudiant::GetAge() const
56:  {
57:     return sonAge;
58:  }
59:
60:  Etudiant& Etudiant::operator=(const Etudiant& rhs)
61:  {
62:     sonNom = rhs.GetNom();
63:     sonAge = rhs.GetAge();
64:     return *this;
65:  }
66:
```

```
 67:  ostream& operator<<(ostream& os, const Etudiant& rhs)
 68:  {
 69:     os << rhs.GetNom() << " a " << rhs.GetAge() << " ans";
 70:     return os;
 71:  }
 72:
 73:  template<class T>
 74:  // affichage des propriétés du vector
 75:  void AffVector(const vector<T>& v);
 76:
 77:  typedef vector<Etudiant> Classe;
 78:
 79:  int main()
 80:  {
 81:     Etudiant Jean;
 82:     Etudiant Michel("Michel", 15);
 83:     Etudiant Martine("Martine", 17);
 84:     Etudiant Pierre("Pierre", 16);
 85:
 86:     Classe ClasseVide;
 87:     cout << "ClasseVide :" << endl;
 88:     AffVector(ClasseVide);
 89:
 90:     Classe ClasseCroissante(3);
 91:     cout << "ClasseCroissante(3) :" << endl;
 92:     AffVector(ClasseCroissante);
 93:
 94:     ClasseCroissante [0] = Jean;
 95:     ClasseCroissante [1] = Michel;
 96:     ClasseCroissante [2] = Martine;
 97:     cout << "ClasseCroissante(3) après ajout des étudiants :"
97a:         << endl;
 98:     AffVector(ClasseCroissante);
 99:
100:    ClasseCroissante.push_back(Pierre);
101:    cout << "ClasseCroissante() après ajout du 4e étudiant :"
101a:       << endl;
102:    AffVector(ClasseCroissante);
103:
104:    ClasseCroissante [0].SetNom("Jean");
105:    ClasseCroissante [0].SetAge(18);
106:    cout << "ClasseCroissante() après SetNom/SetAge :\n";
107:    AffVector(ClasseCroissante);
108:
109:    return 0;
110:  }
111:
```

```
112:  //
113:  // Affichage des propriétés du vector
114:  //
115:  template<class T>
116:  void AffVector(const vector<T>& v)
117:  {
118:     cout << "max_size() = " << v.max_size();
119:     cout << "\tsize() = " << v.size();
120:     cout << "\tcapacity() = " << v.capacity();
121:     cout << "\t" << (v.empty()? "vide": "non vide");
122:     cout << endl;
123:
124:     for (int i = 0; i < v.size(); ++i)
125:        cout << v[i] << endl;
126:
127:     cout << endl;
128:  }
```

Ce programme donne le résultat suivant :

```
ClasseVide :
max_size() = 536870911size() = 0capacity() = 0vide

ClasseCroissante(3) :
max_size() = 536870911size() = 3capacity() = 3non vide
Nouvel étudiant a 16 ans
Nouvel étudiant a 16 ans
Nouvel étudiant a 16 ans

ClasseCroissante(3) après ajout des étudiants :
max_size() = 536870911size() = 3capacity() = 3non vide
Nouvel étudiant a 16 ans
Michel a 15 ans
Martine a 17 ans

ClasseCroissante() après ajout du 4e étudiant :
max_size() = 536870911size() = 4capacity() = 6non vide
Nouvel étudiant a 16 ans
Michel a 15 ans
Martine a 17 ans
Pierre a 16 ans

ClasseCroissante() après SetNom/SetAge :
max_size() = 536870911size() = 4capacity() = 6non vide
Jean a 18 ans
Michel a 15 ans
Martine a 17 ans
Pierre a 16 ans
```

La classe `Etudiant` est définie aux lignes 5 à 23. Les implémentations de ses fonctions membres se situent aux lignes 25 à 65. Pour les raisons que nous avons exposées plus tôt, nous avons défini un constructeur par défaut, un constructeur de copie et un opérateur d'affectation surchargé. Vous pouvez remarquer que sa variable membre `sonNom` est définie comme une instance de la classe `string`. Vous pouvez constater dans ce code qu'il est beaucoup plus facile de travailler avec le type chaîne standard de C++ qu'avec une chaîne `char*` de type C.

La fonction modèle `AffVector()` est déclarée aux lignes 73 et 75 et définie aux lignes 115 à 128. Elle illustre l'utilisation de quelques fonctions membres de `vector` : `max_size()`, `size()`, `capacity()` et `empty()`. En examinant le résultat, vous pouvez constater que le nombre maximal d'objets `Etudiant` qu'un `vector` peut recevoir est 536 870 911 avec notre compilateur ; il pourra être différent avec d'autres types d'éléments. Un `vector` d'entiers, par exemple, peut contenir jusqu'à 1 073 741 823 éléments. Ces valeurs peuvent varier en fonction des compilateurs.

Aux lignes 124 et 125, la valeur de chaque élément du `vector` est affichée *via* l'opérateur d'insertion surchargé `<<`, qui est défini aux lignes 67 à 71.

Les lignes 80 à 85 de la fonction principale du programme crée quatre étudiants. La ligne 86 définit le `vector` vide `ClasseVide` avec le constructeur par défaut de la classe `vector`. Quand un `vector` est créé de cette façon, le compilateur ne réserve pas de mémoire pour lui : comme vous pouvez le constater avec ce que produit l'appel `AffVector(ClasseVide)`, sa taille et sa capacité sont toutes les deux nulles.

La ligne 90 définit un `vector` de trois objets `Etudiant`. Sa taille et sa capacité sont de 3. Les éléments de `ClasseCroissante` sont des objets `Etudiant` qui lui sont affectés aux lignes 94 à 96 *via* l'opérateur d'indexation.

`Pierre`, le quatrième étudiant, est ajouté au `vector` à la ligne 100. La taille du conteneur passe donc à quatre et nous constatons que sa capacité est maintenant de six, ce qui signifie que le compilateur a alloué de la mémoire pour stocker jusqu'à six objets `Etudiant`.

Les `vector` devant être stockés dans un bloc contigu de mémoire, l'augmentation de leur capacité impose une série d'opérations. Un nouveau bloc de mémoire suffisant pour recevoir les quatre objets `Etudiant` doit tout d'abord être alloué. Les trois éléments existants sont ensuite copiés dans ce nouveau bloc et le quatrième est ajouté à la suite du troisième. Pour terminer, le bloc de mémoire initial est libéré. Quand le conteneur contient beaucoup d'éléments, ce processus d'allocation et de libération de mémoire peut prendre du temps ; c'est la raison pour laquelle le compilateur appliquer une stratégie d'optimisation pour réduire la fréquence d'opérations si coûteuses. Dans cet exemple, si nous ajoutons deux objets supplémentaires au `vector`, il ne sera pas nécessaire de libérer et d'allouer à nouveau de la mémoire.

Nous utilisons à nouveau l'opérateur d'indexation au lignes 104 et 105 pour changer les variables membres du premier objet de `ClasseCroissante`.

Faire	*Ne pas faire*
● Définir un constructeur par défaut pour toute classe susceptible d'être stockée dans un `vector`. ● Définir un constructeur de copie pour cette classe. ● Définir un opérateur d'affectation surchargé pour cette classe.	● Créer votre propre classe `vector` ! Vous pouvez utiliser celle de la STL : comme elle fait partie de la norme, tous les compilateurs compatibles devraient en disposer.

La classe conteneur `vector` possède d'autres fonctions membres. La méthode `front()`, en particulier, renvoie une référence vers le premier élément d'une liste alors que `back()` renvoie une référence vers le dernier élément. La méthode `at()` se comporte comme l'opérateur d'indexation `[]`, mais son résultat est plus fiable parce qu'il vérifie si l'indice qui lui a été passé en paramètre se situe bien dans l'intervalle des indices disponibles (vous pourriez, bien sûr, écrire un opérateur d'indexation qui effectue également ces tests). Si l'indice est en dehors des limites, cette méthode lance une exception `out_of_range` (les exceptions sont traitées au prochain chapitre).

La méthode `insert()` insère un ou plusieurs nœuds à une position donnée d'un `vector`. La méthode `pop_back()` supprime le dernier élément. Enfin, `remove()` permet de supprimer un ou plusieurs éléments.

Le conteneur *List*

Une `list` est un conteneur conçu pour être optimal lorsque l'on insére et que l'on supprime fréquemment des éléments. La classe `list` de la STL est définie dans le fichier en tête `<list>` dans l'espace de noms `std`. Elle est implémentée comme une liste doublement chaînée dans laquelle chaque nœud possède des liens à la fois vers le nœud précédent et vers le nœud suivant dans la liste.

La classe `list` dispose de toutes les fonctions membres de la classe `vector`. Généralement, on parcourt une liste en suivant les liens fournis dans chaque nœud, qui sont typiquement implémentés à l'aide de pointeurs. Pour ce type de traitement, la classe conteneur `list` standard utilise un mécanisme nommé *itérateur.*

Un itérateur est une généralisation d'un pointeur qui tente d'éviter certains de ses dangers. En déréférençant un itérateur, on obtient le nœud sur lequel il pointe. Le Listing 19.9 illustre l'utilisation des itérateurs pour l'accès aux nœuds d'une liste.

Listing 19.9 : Consultation d'une liste avec un itérateur

```
 0:  #include <iostream>
 1:  #include <list>
 2:  using namespace std;
 3:
 4:  typedef list<int> ListeEntiers;
 5:
 6:  int main()
 7:  {
 8:     ListeEntiers listeInt;
 9:
10:     for (int i = 1; i <= 10; ++i)
11:        listeInt.push_back(i * 2);
12:
13:     for (ListeEntiers::const_iterator iter = listeInt.begin();
14:                          iter != listeInt.end(); ++iter)
15:        cout << *iter << " ";
16:
17:     return 0;
18:  }
```

L'exécution de ce programme produit le résultat suivant :

```
2 4 6 8 10 12 14 16 18 20
```

Le Listing 19.9 utilise le modèle `list` de la STL. La ligne 1 inclut le fichier en-tête qui contient le code de ce modèle.

La ligne 4 utilise la commande `typedef` pour créer un alias `ListeEntiers` plus facile à lire que `list<int>`.

La ligne 8 définiti l'objet `listeInt` comme une liste d'entiers. Aux lignes 10 et 11, la méthode `push_back()` ajoute à cette liste les dix premiers nombres positifs pairs.

Aux lignes 13 à 15, nous accédons à chaque nœud de la liste à l'aide d'un itérateur constant. Ce choix indique que nous n'avons pas l'intention de modifier les nœuds avec cet itérateur. Si vous voulez pouvoir modifier un nœud désigné par un itérateur, ce dernier doit être un itérateur non constant, c'est-à-dire un `ListeEntiers::iterator`.

La fonction membre `begin()` renvoie un itérateur pointant sur le premier nœud de la liste. Comme on le voit ici, l'opérateur d'incrémentation `++` permet de faire pointer un itérateur sur le nœud suivant. La fonction membre `end()` est assez bizarre : elle renvoie un itérateur pointant sur l'emplacement qui suit le dernier nœud d'une liste. Vous devez donc être certain que votre itérateur n'atteint pas `end()` !

Pour obtenir le nœud pointé, on applique l'opérateur d'indirection à l'itérateur, exactement comme on le fait pour un pointeur (voir la ligne 15).

Bien que nous ayons introduit les itérateurs avec la classe `list`, ils existent également pour la classe `vector`. Outre les méthodes déjà présentées avec cette dernière, la classe `list` fournit également les méthodes `push_front()` et `pop_front()` qui se comportent exactement comme `push_back()` et `pop_back()` sauf, qu'au lieu d'ajouter et de supprimer les éléments à la fin d'une liste, elles les ajoutent et les suppriment en tête de la liste.

Le conteneur *stack*

La *pile* est l'une des structures de données les plus souvent utilisées en programmation. La classe modèle `stack` fournie par la STL est, en fait, implémentée comme une enveloppe de conteneur, non comme une classe conteneur indépendante. Elle est définie dans le fichier en tête `<stack>`, dans l'espace de nom `std`.

Une pile est stockée dans un bloc de mémoire alloué de façon contiguë. Vous ne pouvez accéder qu'à l'élément situé à la fin de la pile et vous ne pouvez supprimer que celui-là. Les conteneurs séquence, et en particulier `vector` et `deque`, présentent des caractéristiques analogues. En fait, tout conteneur séquence qui dispose des opérations `back()`, `push_back()` et `pop_back()` pourrait servir à implémenter une pile. La plupart des autres méthodes de conteneur ne sont pas nécessaires pour ce type de structure.

La classe modèle `stack` de la STL est conçue pour recevoir tout type d'objet. La seule restriction est que tous les éléments doivent être de même type.

Une pile est une structure de type LIFO (*Last In, First Out*, dernier entré, premier sorti). Une analogie classique consiste à comparer une pile à une pile d'assiettes sales dans un restaurant. On charge la pile en ajoutant une assiette sur le dessus et on la décharge en prenant l'assiette située à son sommet.

Par convention, l'extrémité ouverte d'une pile est nommée *sommet* de la pile et les opérations d'ajout et de suppression sont appelées, respectivement, *empiler* et *dépiler*. La classe `stack` hérite de ces termes conventionnels.



La classe `stack` de la STL n'est pas tout à fait identique au mécanisme de pile utilisé par les compilateurs et les systèmes d'exploitation car ces derniers sont capables d'empiler des objets de types différents. La fonctionnalité sous-jacente est, cependant, analogue.

Le conteneur *deque*

Vous pouvez considérer un `deque` comme un `vector` avec deux extrémités : il hérite de l'efficacité de la classe conteneur `vector` pour les opérations de lecture et d'écriture séquentielles, mais la classe conteneur `deque` fournit également des opérations optimisées sur les "extrémités" du conteneur. Ces opérations sont implémentées comme celles de la classe conteneur `list`, avec des allocations de mémoire qui n'auront lieu que pour les nouveaux éléments. Cette caractéristique de la classe `deque` supprime donc le besoin de réallouer tout le conteneur dans un nouvel emplacement mémoire, comme c'est le cas avec la classe `vector`. Les `deque` conviennent donc particulièrement bien aux applications dans lesquelles les insertions et les suppressions s'effectuent surtout en début ou en fin de liste et pour lesquelles il est aussi important de pouvoir accéder séquentiellement aux éléments. Un simulateur d'assemblage de convoi ferroviaire en est un bon exemple puisque les wagons peuvent être ajoutés des deux côtés d'un train.

Le conteneur *queue*

Une *file attente* est une autre structure de données très utilisée en programmation. Les éléments sont ajoutés dans la file à partir d'une extrémité puis récupérés à l'autre extrémité.

Une file d'attente peut être comparée à celle d'un théâtre. On entre dans la file en se positionnant à la fin, et on la quitte lorsqu'on est en tête. C'est donc une structure du type FIFO (*First In, First Out* pour premier entré, premier sorti) alors qu'une pile est une structure de type LIFO.

Comme `stack`, `queue` est implémentée sous la forme d'une enveloppe de conteneur. Ce conteneur doit disposer des opérations `front()`, `back()`, `push_back()` et `pop_front()`.

Les conteneurs associatifs

Le conteneur `vector` étudié précédemment est comparable à une version améliorée d'un tableau. Il possède en effet toutes les caractéristiques des tableaux, plus quelques fonctions supplémentaires. Il souffre malheureusement également d'un inconvénient significatif des tableaux : vous ne pouvez pas retrouver un élément autrement que par son indice dans le conteneur. Les conteneurs associatifs, en revanche, offrent un accès direct rapide, reposant sur une association entre des *clés* et des *valeurs*.

Les conteneurs de séquence sont conçus pour un accès séquentiel et direct aux éléments à l'aide d'un indice ou d'un itérateur ; les conteneurs associatifs sont conçus pour un accès direct rapide aux éléments à l'aide de clés. La bibliothèque standard de C++ fournit cinq conteneurs associatifs : `map`, `multimap`, `set`, `multiset` et `bitset`.

Le conteneur *map*

Le premier conteneur associatif que nous présenterons s'appelle `map`. Son nom provient de l'idée qu'il contient des "cartes" (*maps*), c'est-à-dire la clé vers la valeur associée, exactement comme un point sur une carte géographique correspond à un véritable endroit sur la Terre. Dans le Listing 19.10, on utilise un `map` pour implémenter l'exemple de la classe d'étudiants déjà présenté dans le Listing 19.8.

Listing 19.10 : Classe conteneur *map*

```
 0:  #include <iostream>
 1:  #include <string>
 2:  #include <map>
 3:  using namespace std;
 4:
 5:  class Etudiant
 6:  {
 7:  public:
 8:     Etudiant();
 9:     Etudiant(const string& nom, const int age);
10:     Etudiant(const Etudiant& rhs);
11:     ~Etudiant();
12:
13:     void   SetNom(const string& nom);
14:     string GetNom() const;
15:     void   SetAge(const int age);
16:     int    GetAge()   const;
17:
18:     Etudiant& operator=(const Etudiant& rhs);
19:
20:  private:
21:     string sonNom;
22:     int sonAge;
23:  };
24:
25:  Etudiant::Etudiant() :
26:    sonNom("Nouvel étudiant"), sonAge(16)
27:  {}
28:
29:  Etudiant::Etudiant(const string& nom, const int age) :
30:    sonNom(nom), sonAge(age)
31:  {}
32:
33:  Etudiant::Etudiant(const Etudiant& rhs) :
34:    sonNom(rhs.GetNom()), sonAge(rhs.GetAge())
35:  {}
```

```
36:
37:  Etudiant::~Etudiant()
38:  {}
39:
40:  void Etudiant::SetNom(const string& nom)
41:  {
42:     sonNom = nom;
43:  }
44:
45:  string Etudiant::GetNom() const
46:  {
47:     return sonNom;
48:  }
49:
50:  void Etudiant::SetAge(const int age)
51:  {
52:     sonAge = age;
53:  }
54:
55:  int Etudiant::GetAge() const
56:  {
57:     return sonAge;
58:  }
59:
60:  Etudiant& Etudiant::operator=(const Etudiant& rhs)
61:  {
62:     sonNom = rhs.GetNom();
63:     sonAge = rhs.GetAge();
64:     return *this;
65:  }
66:
67:  ostream& operator<<(ostream& os, const Etudiant& rhs)
68:  {
69:     os << rhs.GetNom() << " a " << rhs.GetAge() << " ans";
70:     return os;
71:  }
72:
73:  template<class T, class A>
74:  void AffMap(const map<T, A>& v);
75:
76:  typedef map<string, Etudiant> Classe;
77:
78:  int main()
79:  {
80:     Etudiant Jean("Jean", 18);
81:     Etudiant Martine("Martine", 15);
82:     Etudiant Michel("Michel", 17);
```

```
 83:     Etudiant Pierre("Pierre", 16);
 84:
 85:     Classe ClasseMaths;
 86:     ClasseMaths [Jean.GetNom()] = Jean;
 87:     ClasseMaths [Martine.GetNom()] = Martine;
 88:     ClasseMaths [Michel.GetNom()]  = Michel;
 89:     ClasseMaths [Pierre.GetNom()] = Pierre;
 90:
 91:     cout << "ClasseMaths :" << endl;
 92:     AffMap(ClasseMaths);
 93:
 94:     cout << "On sait que " << ClasseMaths ["Michel"].GetNom()
 95:         << " a " << ClasseMaths["Michel"].GetAge()
 96:         << " ans" << endl;
 97:     return 0;
 98: }
 99:
100: //
101: // Affiche les propriétés du map
102: //
103: template<class T, class A>
104: void AffMap(const map<T, A>& v)
105: {
106:    for (map<T, A>::const_iterator iter = v.begin();
107:                 iter != v.end(); ++iter)
108:       cout << iter->first << " : " << iter->second << endl;
109:
110:    cout << endl;
111: }
```

Voici le résultat de l'exécution de ce programme :

```
ClasseMaths :
Michel : Michel a 17 ans
Jean : Jean a 18 ans
Pierre : Pierre a 16 ans
Martine : Martine a 15 ans

On sait que Michel a 17 ans
```

Dans cet exemple, on crée une classe de Maths et on y ajoute quatre étudiants. La liste est ensuite affichée. Puis, Michel est affiché avec son âge mais, au lieu d'utiliser un indexeur numérique, comme dans l'exemple précédent, c'est le nom de Michel qui sert à retrouver son âge. Ceci est possible grâce au modèle `map`.

Vous pouvez constater également que la majeure partie du listing correspond à la classe `Etudiant`. C'est un code que vous devriez maintenant comprendre.

Les seuls éléments propres à ce listing démarrent à la ligne 2, où le fichier en tête `<map>` est inclus parce que nous allons utiliser le conteneur `map` standard. À la ligne 73, vous pouvez constater que l'on fournit un prototype pour la fonction `AffMap()`, qui est une fonction modèle. Elle permet d'afficher les éléments du `map`.

La ligne 76 utilise `typedef` pour définir `Classe` comme un `map` dont les éléments sont constitués de paires (clé, valeur). Le premier élément d'une paire correspond à une chaîne qui est la valeur de la clé. Dans notre `Classe`, nous utilisons le nom des étudiants comme valeur de clé ; cette dernière est donc de type `string`. La clé de chaque élément du conteneur doit être unique, il ne doit donc pas avoir deux éléments avec la même clé. Le second élément de la paire représente l'objet réel, ici un objet `Etudiant`. Le type des données "paire" est implémenté dans la STL sous la forme d'une `struct` de deux membres : `first` et `second`. Nous pouvons utiliser ces membres pour accéder à la clé et à la valeur d'un nœud.

Commençons par étudier la fonction `AffMap()` définie de la ligne 103 à la ligne 111. Celle-ci utilise un itérateur constant pour accéder à un objet du map. À la ligne 108, `iter->first` pointe sur la clé, c'est-à-dire sur le nom d'un étudiant et `iter->second` pointe sur l'objet `Etudiant`.

Il ne reste que la fonction `main()` qui est comprise entre les lignes 78 et 98. Intéressons-nous aux lignes 80 à 83, dans lesquelles quatre objets `Etudiant` sont créés. La `ClasseMaths` est définie comme une instance de notre `Classe` de la ligne 85. Au lignes 86 à 89, nous ajoutons les quatre étudiants de la `ClasseMaths` en utilisant la syntaxe suivante :

```
objet_map[clé] = valeur_objet;
```

À la ligne 86, vous pouvez constater que la `clé` utilisée correspond au nom d'un objet `Etudiant`, obtenu grâce à sa méthode `GetNom()`. `valeur_objet` est un objet `Etudiant`.

Vous pourriez également utiliser les fonctions `push_back()` ou `insert()` pour ajouter une paire (clé, valeur) dans le `map`. Vous trouverez des informations complémentaires à ce sujet en consultant la documentation de votre compilateur.

Lorsque tous les objets `Etudiant` ont été ajoutés dans le map, vous pouvez accéder à l'un d'entre eux par sa clé. Aux lignes 94 à 96, `ClasseMaths["Michel"]` permet donc d'obtenir l'enregistrement de Michel.

Les autres conteneurs associatifs

La classe conteneur `multimap` est une classe `map` dans laquelle la contrainte des clés uniques n'existe pas. Plusieurs éléments peuvent donc avoir la même clé.

La classe conteneur `set` ressemble également à la classe `map`. La seule différence est que ses éléments ne sont pas des paires (clé, valeur). Un élément est uniquement constitué de

la clé. La classe conteneur `multiset` est une classe `set` dans laquelle les clés dupliquées sont autorisées.

Enfin, la classe conteneur `bitset` est un modèle qui sert à stocker une suite de bits.

Les classes d'algorithmes

Un conteneur est un emplacement idéal pour stocker une séquence d'éléments. Tous les conteneurs standard définissent des opérations qui permettent de manipuler les conteneurs et leurs éléments. Implémenter toutes ces opérations dans vos propres séquences peut cependant se révéler laborieux et représenter une source d'erreurs importante. Ces opérations étant souvent les mêmes sur la plupart des séquences, un ensemble d'algorithmes génériques permet d'éviter de devoir écrire ses propres opérations pour chaque nouveau conteneur. La bibliothèque standard fournit ainsi une soixantaine d'algorithmes standard qui réalisent les opérations les plus courantes et les plus basiques sur les conteneurs.

Les algorithmes standard sont définis dans `<algorithm>`, dans l'espace de noms `std`.

Pour comprendre le fonctionnement d'un algorithme, vous devez comprendre le concept d'objet fonction. Un *objet fonction* est une instance d'une classe qui définit l'opérateur `()` surchargé : il peut donc être appelé comme une fonction, comme le montre le e Listing 19.11.

Listing 19.11 : Un objet fonction

```
 0:  #include <iostream>
 1:  using namespace std;
 2:
 3:  template<class T>
 4:  class Print    {
 5:  {
 6:     public:
 7:       void operator()(const T& t)
 8:      {
 9:       cout << t << " ";
10:     }
11:  };
12:
13:  int main()
14:  {
15:      Print<int> DoPrint;
16:      for (int i = 0; i < 5; ++i)
17:         DoPrint(i);
18:      return 0;
19:  }
```

Voici le résultat de l'exécution du programme :

```
0 1 2 3 4
```

La classe modèle `Print` est définie aux lignes 3 à 11 ; il s'agit d'une classe modèle standard. Aux lignes 6 à 9, l'opérateur `()` est surchargé pour recevoir un objet en paramètre et l'afficher sur la sortie standard. La ligne 15 définit `DoPrint` comme une instance de la classe `Print` utilisant une valeur `int`. Nous pouvons ensuite l'utiliser exactement comme une fonction pour afficher n'importe quelle valeur entière, comme le montre la ligne 17. Les classes d'algorithme standard fonctionnent exactement comme `Print` : elles ont surchargé l'opérateur `()` pour que vous puissiez les utiliser comme des fonctions.

Les opérations sur les séquences sans modification

Les opérations sur les séquences sans modification sont des composants de la bibliothèque d'algorithmes qui réalisent des opérations sur une séquence sans en modifier les éléments. Il s'agit d'opérations comme `for_each()`, `find()`, `search()` et `count()`. Le Listing 19.12 montre comment utiliser un objet fonction et l'algorithme `for_each` pour afficher les éléments d'un `vector`.

Listing 19.12 : Algorithme *for_each()*

```
 0:  #include <iostream>
 1:  #include <vector>
 2:  #include <algorithm>
 3:  using namespace std;
 4:
 5:  template<class T>
 6:  class Print
 7:  {
 8:  public:
 9:     void operator()(const T& t)
10:     {
11:        cout << t << " ";
12:     }
13:  };
14:
15:  int main()
16:  {
17:     Print<int>  DoPrint;
18:     vector<int> vInt(5);
19:
20:     for (int i = 0; i < 5; ++i)
21:        vInt[i] = i * 3;
22:
```

```
23:     cout << "for_each()" << endl;
24:     for_each(vInt.begin(), vInt.end(), DoPrint);
25:     cout << endl;
26:
27:     return 0;
28:  }
```

L'exécution du programme donne le résultat suivant :

```
for_each()
0 3 6 9 12
```

Tous les algorithmes standard de C++ sont définis dans le fichier `<algorithm>`, que nous avons donc inclus à la ligne 2. Même si la plus grande partie du programme nécessite peu d'explications, une ligne mérite notre attention. La fonction `for_each()` est appelée à la ligne 24 pour parcourir chaque élément du `vector vInt`. Pour chaque élément de ce `vector`, elle appelle l'objet fonction `DoPrint` et passe en paramètre l'élément à `DoPrint.operator()`. La valeur de cet élément est donc affichée sur l'écran.

Les opérations de séquence avec modification

Les opérations de séquence avec modification réalisent des opérations qui modifient les éléments de la séquence, comme les opérations de remplissage ou de réorganisation des collections. Le Listing 19.13 illustre l'algorithme `fill()`.

Listing 19.13 : Algorithme de séquence avec modification

```
 0:  #include <iostream>
 1:  #include <vector>
 2:  #include <algorithm>
 3:  using namespace std;
 4:
 5:  template<class T>
 6:  class Print
 7:  {
 8:  public:
 9:     void operator()(const T& t)
10:     {
11:        cout << t << " ";
12:     }
13:  };
14:
15:  int main()
16:  {
17:     Print<int>   DoPrint;
```

```
18:     vector<int>  vInt(10);
19:
20:     fill(vInt.begin(), vInt.begin() + 5, 1);
21:     fill(vInt.begin() + 5, vInt.end(), 2);
22:
23:     for_each(vInt.begin(), vInt.end(), DoPrint);
24:     cout << endl << endl;
25:
26:     return 0;
27: }
```

Ce listing donne le résultat suivant :

```
1 1 1 1 1 2 2 2 2 2
```

La seule partie nouvelle dans ce listing se situe aux lignes 20 et 21, dans lesquelles apparaît l'algorithme `fill()`. Ce dernier stocke des éléments dans une séquence avec une valeur donnée. À la ligne 20, il affecte la valeur entière 1 au cinq premiers éléments de `vInt`. Puis, il affecte 2 aux cinq derniers éléments du conteneur (ligne 21).

Tri et opérations apparentées

La troisième catégorie d'algorithmes concerne le tri et les opérations apparentées. Dans cet ensemble d'opérations, se trouvent la fusion, les tris partiels, les tris partiels avec copie, les recherches binaires, les vérifications des limites inférieure et supérieure, les intersections d'ensembles, les différences d'ensembles, les minimums, les maximums, les permutations, etc. Consultez la documentation du compilateur ou celle de la norme C++ pour en savoir plus sur chacune de ces opérations.



Cet ouvrage n'a pas pour objet de voir en détail toutes les opérations des classes d'algorithmes et de la STL. Vous pouvez consulter la documentation de votre compilateur ou la norme C++ pour en savoir plus sur les classes et les opérations disponibles, ainsi que sur leurs paramètres et leur utilisation. En outre, il existe des livres consacrés entièrement à la STL.

Questions-réponses

Q Pourquoi utiliser des modèles plutôt que des macros ?

R Les modèles permettent de contrôler le type et sont intégrés au langage. Ils peuvent ainsi être vérifiés par le compilateur, au moins lorsque vous instanciez la classe pour créer une variable particulière.

Q Quelle différence y a-t-il entre le type paramétré d'une fonction modèle et les paramètres d'une fonction normale ?

R Une fonction normale (qui n'est pas modèle) reçoit des paramètres sur lesquels elle peut agir. Avec une fonction modèle, vous pouvez paramétrer le type d'un paramètre particulier de la fonction. Cela signifie que vous pouvez transmettre un tableau de `Type` à une fonction : ce `Type` sera déterminé par la définition de la variable qui est une instance de la classe pour un type spécifique.

Q Comment choisir entre l'utilisation des modèles et celle de l'héritage ?

R Utilisez les modèles lorsque la seule chose qui varie est le type de l'élément sur lequel la classe agit. Si vous devez recopier une classe existante pour n'en changer que le type d'un ou plusieurs de ses membres, un modèle sera probablement une bonne solution. Employez aussi un modèle lorsque vous êtes tenté de modifier une classe pour opérer sur une classe ancêtre de plus haut niveau (ce qui réduit la sécurité de type) de ses opérandes ou pour amener deux classes non liées à partager un ancêtre commun, de sorte que votre classe puisse fonctionner avec les deux (ce qui réduit une fois de plus la sécurité de type).

Q Quand faut-il utiliser les classes amies modèles générales?

R Lorsque chaque instance, quel que soit le type, doit être une amie de cette classe ou de cette fonction.

Q Quand faut-il utiliser les classes ou fonctions amies modèles spécifiques ?

R Lorsque vous voulez mettre deux classes en relation. Par exemple, `Array<int>` devra correspondre à `iterator<int>` et non à `iterator<Animal>`.

Testez vos connaissances

1. Quelle est la différence entre un modèle et une macro ?
2. Quelle est la différence entre le paramètre d'un modèle et celui d'une fonction ?
3. Quelle est la différence entre une classe amie modèle spécifique et une classe amie modèle générale ?
4. Peut-on fournir un comportement particulier à une instance d'un modèle, mais pas aux autres ?
5. Combien de variables statiques sont créées quand vous placez un membre statique dans la définition d'une classe modèle ?
6. Quels attributs doit avoir votre classe pour être utilisée avec les conteneurs standard ?
7. Que signifie STL et pourquoi est-elle importante ?

Exercices

1. Créez un modèle basé sur cette classe `Liste` :

```
class Liste
{
private:

public:
     Liste():tete(0),fin(0),leNbre(0) {}
     virtual ~Liste();
     void insere( int valeur );
     void ajoute( int valeur );
     int present( int valeur ) const;
     int vide() const { return tete == 0; }
     int nbre() const { return leNbre; }
private:
     class ListeCellule
     {
     public:
       ListeCellule( int valeur, ListCell *cell = 0):val(valeur),
                    suivante(cell){}
       int val;
       ListeCellule *suivante;
     };
     ListeCellule *tete;
     ListeCellule *fin;
     int leNbre;
};
```

2. Écrivez l'implémentation de la version non modèle de cette classe `Liste`.
3. Écrivez la version modèle des implémentations.
4. Déclarez trois objets `Liste` : une liste de chaînes, une liste de chats et une liste d'entiers.
5. CHERCHEZ L'ERREUR : (on suppose que le modèle `Liste` est défini et que `Chat` est la classe qui a été définie dans les exemples précédents).

```
Liste<Chat> ListeChats;
Chat Felix;
ListeChats.ajoute( Felix );
cout << "Felix est "
     << (ListeChats.present(Felix) ) ? "présent" : "absent "
     << endl;
```

Remarque : qu'est-ce qui différencie `Chat` de `int` ?

6. Déclarez l'opérateur ami == pour `Liste`.
7. Implémentez l'opérateur ami == pour `Liste`.
8. `operator==` présente-t-il le même problème que celui de l'Exercice 5 ?
9. Implémentez une fonction modèle pour `permuter` qui échange deux variables.



20

Gestion des erreurs et exceptions

Au sommaire de ce chapitre

- Les exceptions
- L'utilisation des exceptions et les problèmes qu'elles soulèvent
- La construction des hiérarchies d'exceptions
- La place des exceptions dans une approche générale de gestion des erreurs
- Les débogueurs

Les lignes de code que vous avez étudiées jusqu'à présent n'étaient que des exemples : elles ne comprenaient donc aucun traitement des erreurs afin de se concentrer sur les problèmes qu'elles avaient pour but d'illustrer. Les vrais programmes, par contre, doivent prendre en considération les conditions d'erreurs.

Erreurs logiques et fautes de syntaxe

Il est rare que les programmes réels ne contiennent pas au moins une erreur ou un bogue. Plus un programme est long, plus la probabilité des bogues est importante. En fait, dans les gros programmes, la plupart de ces bogues sont éliminés dans la version finale du produit, mais cela ne doit pas pour autant nous satisfaire. La création de programmes robustes et sans bogues doit être la première priorité de tout programmeur sérieux.

Les programmes bogués et instables sont le problème le plus important de l'industrie du logiciel. Les tests et la correction du code représentent l'une des plus grosses dépenses dans un projet de développement. Celui qui trouvera la solution pour produire des programmes fiables à faible coût révolutionnera le monde du logiciel.

Un programme peut contenir divers types d'erreurs. Les plus courantes sont les erreurs de logique ou de syntaxe. La première se produit lorsque vous n'avez pas conçu correctement les algorithmes du programme, la seconde lorsque vous n'utilisez pas la bonne expression, fonction ou structure.

La recherche et l'expérience dans le monde réel ont démontré que plus un problème logique est identifié tard dans le processus de développement, plus sa correction est coûteuse. Les problèmes et les erreurs les moins chers à corriger sont ceux que vous avez évité de créer ou que le compilateur a repérés. Les normes C++ poussent les compilateurs à identifier de plus en plus d'erreurs au moment de la compilation.

La correction des erreurs qui passent le cap du compilateur et que l'on détecte au cours des premiers tests (celles qui apparaissent à tous les coups) est moins coûteuse que celle des erreurs qui se produisent de façon intermittente.

Un problème plus important que les erreurs de logique ou de syntaxe concerne la *fragilité* : votre programme se comporte très bien tant que l'utilisateur entre un nombre lorsqu'on le lui demande, mais ne marche plus dès que l'utilisateur entre une lettre, par exemple. D'autres programmes s'interrompent s'ils manquent de mémoire, si la disquette ne se trouve pas dans le lecteur ou si la connexion Internet est perdue.

Les programmeurs font de gros efforts pour éliminer ce type de comportement. Ils essaient d'obtenir des programmes "blindés" pouvant gérer toutes sortes de situations qui pourraient se produire pendant l'exécution, des saisies indélicates au manque de mémoire.



Il est important de différencier les erreurs qui sont dues à une faute de syntaxe du programmeur, les erreurs logiques qui sont la conséquence d'une mauvaise compréhension ou résolution d'un problème de la part du programmeur et les exceptions qui se produisent en cas de problème inhabituel, mais prévisible, comme un manque de ressource (mémoire ou disque).

Circonstances exceptionnelles

Vous ne pouvez pas empêcher les situations exceptionnelles, vous devez simplement vous y préparer. Que se passera-t-il si votre programme exige de la mémoire pour allouer dynamiquement un objet et qu'il n'y en a plus de disponible ? Comment répondra-t-il ? Que faire si vous produisez l'une des erreurs mathématiques les plus classique en divisant par zéro ? Voici quelques possibilités :

- interrompre brutalement le programme ;
- informer l'utilisateur puis arrêter proprement le programme ;
- informer l'utilisateur et lui permettre d'essayer de corriger et de poursuivre ;
- prendre les mesures qui s'imposent et poursuivre l'exécution du programme sans inquiéter l'utilisateur.

Le Listing 20.1 est extrêmement simple et prêt à se planter, mais il illustre un problème très sérieux, présent dans de nombreux programmes !

Listing 20.1 : Créer une situation exceptionnelle

```
 0:  // Ce programme va planter
 1:  #include <iostream>
 2:  using namespace std;
 3:
 4:  const int TailleDefaut = 10;
 5:
 6:  int main()
 7:  {
 8:     int haut = 90;
 9:     int bas  = 0;
10:
11:     cout << "haut / 2 = " << (haut/ 2) << endl;
12:
13:     cout << "haut divisé par bas = ";
14:     cout << (haut / bas) << endl;
15:
16:     cout << "haut / 3 = " << (haut/ 3) << endl;
17:
18:     cout << "Terminé." << endl;
19:     return 0;
20: }
```

Voici le résultat de l'exécution du programme :

```
haut / 2 = 45
haut divisé par bas =
```



Ce programme peut afficher ce résultat sur la console, mais il plantera sûrement immédiatement après.

Le Listing 20.1 a été conçu pour planter ; si vous aviez demandé à l'utilisateur d'entrer deux nombres, il aurait pu produire le même résultat.

Les lignes 8 et 9 déclarent et initialisent deux variables entières. Vous auriez également pu demander ces deux nombres à l'utilisateur ou les lire à partir d'un fichier. Aux lignes 11, 14 et 16, ces nombres sont utilisés dans des opérations mathématiques et plus particulièrement pour effectuer une division. Si les lignes 11 et 16 ne posent pas de problème, la ligne 14 en présente. La division par zéro pose un problème exceptionnel, un plantage. Le programme se termine et une exception sera sûrement affichée par le système d'exploitation.

Bien qu'il ne soit pas toujours nécessaire (ni même souhaitable) que tous les programmes gèrent automatiquement toutes sortes de situations, il est clair qu'une simple interruption de ces programmes est insuffisante.

La gestion des exceptions fournies par C++ permet de traiter toutes les conditions inhabituelles, mais prévisibles, qui peuvent se produire pendant l'exécution d'un programme.

Les coulisses des exceptions

Le principe des exceptions est très simple :

- L'ordinateur tente d'exécuter un élément de code qui pourrait tenter d'allouer des ressources comme de la mémoire, essayer de verrouiller un fichier ou effectuer tout type de tâche.
- On intègre la logique (le code) au cas où les opérations que vous essayez d'exécuter échoueraient pour une raison exceptionnelle. Vous pourriez, par exemple, inclure du code pour intercepter tous les problèmes (la mémoire ne peut être allouée, le fichier ne peut être verrouillé, etc.).
- Si le code est utilisé par un autre code (par une fonction, par exemple), on a également besoin d'un mécanisme permettant de faire remonter les informations sur les problèmes (exceptions) de votre niveau au niveau précédent. Il devrait exister un chemin allant du code qui a posé problème jusqu'à celui qui traite l'erreur. Si ces deux extrémités sont séparées par des couches de fonctions, celles-ci devraient avoir la possibilité de résoudre le problème, mais ne devraient pas être obligées d'inclure du code dans le seul but de transmettre la condition d'erreur.

La gestion des exceptions associe ces trois points, de manière relativement simple.

Les éléments de la gestion des exceptions

Pour gérer les exceptions, vous devez d'abord identifier la partie du code qui doit être surveillée. Pour cela, on utilise un bloc `try`.

Tout code susceptible de poser problème devrait être placé dans un bloc `try`, dont le format de base est le suivant :

```
try
{
UneFonctionDangereuse();
}
catch (...)
{
}
```

Ici, si une exception survient lors de l'exécution de `UneFonctionDangereuse()`, elle est notée et interceptée. Il suffit d'ajouter le mot-clé `try` et les accolades pour que le programme commence à surveiller les exceptions. Bien sûr, en cas d'exception, il faut agir.

Si une exception survient au cours de l'exécution du code contenu dans un bloc `try`, on dit que l'exception est "lancée". Elle peut ensuite être interceptée (capturée) avec un bloc `catch`. Lorsqu'une exception est lancée, le contrôle est transmis au bloc `catch` approprié qui suit le bloc `try`. Dans l'exemple précédent, l'ellipse `(...)` désigne n'importe quelle exception, mais on peut également capturer des types spécifiques d'exceptions. Pour cela, on utilise un ou plusieurs blocs `catch` après le bloc `try`, comme dans l'exemple suivant :

```
try
{
UneFonctionDangereuse();
}
catch(OutOfMemory)
{
// actions
}
catch(FileNotFound)
{
// autres actions
}
catch (...)
{
}
```

Cet exemple *traite* les exceptions qui pourraient être lancées par `UneFonctionDangereuse()`. Si une exception est lancée, elle est envoyée dans le premier bloc `catch` qui suit

immédiatement le bloc `try`. Si celui-ci possède un paramètre de type, comme c'est le cas dans cet exemple, l'exception est contrôlée pour voir si elle correspond au type indiqué. Sinon, on vérifie la prochaine instruction `catch`, etc., jusqu'à trouver une concordance ou autre chose qu'un bloc `catch`. À la première concordance, le bloc `catch` correspondant est exécuté. À moins de vouloir vraiment laisser passer les autres types d'exceptions, il est donc toujours préférable que le dernier bloc `catch` utilise l'ellipse.



Un bloc `catch` est aussi appelé gestionnaire d'exception, car il permet de gérer une exception.



Vous pouvez considérer les blocs `catch` comme des fonctions surchargées. En cas de concordance de la signature, cette fonction est exécutée.

Voici les étapes de base pour le traitement des exceptions :

1. Identifiez les zones du programme dans lesquelles une opération peut conduire à une situation exceptionnelle et mettez-les dans des blocs `try`.
2. Créez des blocs `catch` pour capturer les exceptions si elles sont lancées. Vous pouvez créer un bloc `catch` pour un type d'exception spécifique (en précisant un paramètre typé pour le bloc `catch`) ou pour toutes les exceptions (en utilisant une ellipse `(...)` comme paramètre).

Le Listing 20.2 ajoute une gestion minimale des exceptions au Listing 20.1. Il utilise un bloc `try` et un bloc `catch`.



Certains compilateurs très anciens ne supportent pas les exceptions. Si votre compilateur en fait partie, vous devrez installer sa version la plus récente si vous voulez compiler les exercices de ce chapitre. Les exceptions faisant partie de la norme ANSI C++, elles sont bien sûr supportées par toutes les versions actuelles des compilateurs.

Listing 20.2 : Intercepter une exception

```
0:  // tentative d'interception
1:  #include <iostream>
2:  using namespace std;
3:
4:  const int TailleDefaut = 10;
5:
6:  int main()
7:  {
8:     int haut = 90;
```

```
 9:     int bas  = 0;
10:
11:     try
12:     {
13:        cout << "haut / 2 = " << (haut/ 2) << endl;
14:
15:        cout << "haut divisé par bas = ";
16:        cout << (haut / bas) << endl;
17:
18:        cout << "haut / 3 = " << (haut/ 3) << endl;
19:     }
20:     catch(...)
21:     {
22:        cout << "Il y a un problème !" << endl;
23:     }
24:
25:     cout << "Terminé." << endl;
26:     return 0;
27:  }
```

Voici le résultat de l'exécution du programme :

```
haut / 2 = 45
haut divise par bas = Il y a un problème !
Terminé.
```

À la différence du listing précédent, l'exécution du Listing 20.2 ne plante pas. Le programme est désormais capable de signaleur une erreur et de se terminer proprement.

Nous avons ajouté un bloc `try` autour du code où aurait pu se produire un problème. Ici, c'est autour de la division (lignes 11 à 19). Au cas où une exception surviendrait, nous avons fait suivre ce bloc `try` d'un bloc `catch`, aux lignes 20 à 23.

Ce bloc `catch` de la ligne 20 contient trois points, ou ellipse. Comme on l'a déjà indiqué, c'est un cas spécial qui indique que toutes les exceptions survenant dans le code `try` précédent seront traitées par ce bloc, à moins qu'un bloc `catch` précédent ne l'ait fait. Dans ce listing, ce sera sûrement une erreur de division par zéro. Comme vous le verrez plus loin, il est souvent préférable de rechercher des types d'exceptions plus spécifiques, afin de pouvoir personnaliser la gestion de chacune.

Vous remarquerez que ce listing ne plante pas. En outre, le résultat vous montre que son exécution s'est poursuivie à la ligne 25, après l'instruction `catch`. Cela est confirmé par l'affichage du mot `Terminé`.

Blocs `try`

Un bloc `try` est une suite d'instructions commençant par le mot-clé `try`, suivi d'une accolade ouvrante et se terminant par une accolade fermante.

Exemple :

```
try
{
   Fonction();
};
```

Blocs `catch`

Un bloc `catch` est un élément de code commençant par le mot-clé `catch`, suivi d'un type d'exception entre parenthèses, puis d'une accolade ouvrante et se terminant par une accolade fermante. Les blocs `catch` ne sont autorisés qu'après un bloc `try`.

Exemple :

```
try
{
   Fonction();
};
catch (OutOfMemory)
{
   // action
}
```

Produire ses propres exceptions

Le Listing 20.2 a montré deux des aspects de la gestion des exceptions – marquer le code à surveiller et préciser comment l'exception doit être traitée. Cependant, il ne traitait que les exceptions prédéfinies. La troisième partie du traitement des exceptions consiste à pouvoir à créer ses propres types d'exceptions. En créant vos propres exceptions, vous êtes en mesure de disposer de gestionnaires personnalisés (blocs `catch`) pour les exceptions spécifiques à votre application.

Pour créer une exception amenant l'instruction `try` à réagir, on utilise le mot-clé `throw` (*lancer*). En gros, on lance l'exception et on peut espérer qu'un gestionnaire (bloc `catch`) l'intercepte. Le format de base de l'instruction `throw` est le suivant :

```
throw exception;
```

Cette instruction lance *exception*. Le contrôle est alors passé à un gestionnaire ; s'il n'y en a pas, le programme se termine.

La valeur lancée dans l'exception peut être de quasiment n'importe quel type. Comme on l'a indiqué, vous pouvez configurer des gestionnaires pour chaque type d'objet que votre programme pourrait déclencher. Le Listing 20.3 montre comment lancer une exception de base en modifiant le Listing 20.2.

Listing 20.3 : Lancement d'une exception

```
 0:  // Lancement d'une exception
 1:  #include <iostream>
 2:
 3:  using namespace std;
 4:
 5:  const int TailleDefaut = 10;
 6:
 7:  int main()
 8:  {
 9:     int haut = 90;
10:     int bas  = 0;
11:
12:     try
13:     {
14:       cout << "haut / 2 = " << (haut/ 2) << endl;
15:
16:        cout << "haut divisé par bas = ";
17:        if ( bas == 0 )
18:           throw "Division par zéro !";
19:
20:        cout << (haut / bas) << endl;
21:
22:        cout << "haut / 3 = " << (haut/ 3) << endl;
23:     }
24:     catch( const char * ex )
25:     {
26:        cout << "\n*** " << ex << " ***" << endl;
27:     }
28:
29:     cout << "Terminé." << endl;
30:     return 0;
31:  }
```

Voici le résultat de l'exécution du programme :

```
haut / 2 = 45
haut divisé par bas = *** Division par zéro ! ***
Terminé.
```

À la différence du listing précédent, celui-ci contrôle mieux ses exceptions. Même s'il ne fait pas le meilleur usage de ces exceptions, il montre bien l'utilisation de l'instruction `throw`.

La ligne 17 détermine si la valeur de `bas` est égale à zéro, auquel cas on lance une exception. Ici, cette exception est une chaîne.

L'instruction `catch` de la ligne 24 débute un gestionnaire qui attend un pointeur vers un caractère constant. Avec les exceptions, les chaînes sont capturées par un pointeur de caractère constant : le gestionnaire commençant à la ligne 24 intercepte donc le `throw` de la ligne 18. La ligne 26 affiche entre astérisques la chaîne qui a été passée. La ligne 27 contient l'accolade fermante qui signale la fin du gestionnaire ; le contrôle passe alors à la première ligne qui suit les instructions `catch` et le programme se poursuit jusqu'à la fin.

Si votre exception avait concerné un problème plus sérieux, vous auriez pu quitter l'application après avoir affiché le message de la ligne 26. Si vous lancez l'exception dans une fonction qui a été appelée par une autre fonction, vous pouvez la faire remonter à la fonction appelante. Pour cela, il suffit d'appeler l'instruction `throw` sans aucun paramètre. L'exception en cours sera alors relancée à partir de l'emplacement courant.

Créer une classe d'exception

Vous pouvez créer des classes bien plus complexes pour lancer une exception. Le Listing 20.4 présente une classe `Array` simplifiée, fondée sur le modèle développé au Chapitre 19.

Listing 20.4 : Déclencher une exception

```
 0:  #include <iostream>
 1:  using namespace std;
 2:
 3:  const int TailleDefaut = 10;
 4:
 5:  class Array
 6:  {
 7:  public:
 8:      // constructeurs
 9:      Array(int taille = TailleDefaut);
10:     Array(const Array &rhs);
11:     ~Array() { delete [] pType;}
12:
```

```
13:     // opérateurs
14:     Array& operator=(const Array&);
15:     int& operator[](int indice);
16:     const int& operator[](int indice) const;
17:
18:     // méthode d'accès
19:     int GetTaille() const { return saTaille; }
20:
21:     // fonction amie
22:     friend ostream& operator<< (ostream&, const Array&);
23:
24:     class xLimites {};  // définit la classe exception
25:
26:  private:
27:     int *pType;
28:     int  saTaille;
29:  };
30:
31:  Array::Array(int taille):
32:     saTaille(taille)
33:  {
34:     pType = new int[taille];
35:     for (int i = 0; i < taille; i++)
36:        pType[i] = 0;
37:  }
38:
39:  Array& Array::operator=(const Array &rhs)
40:  {
41:     if (this == &rhs)
42:        return *this;
43:     delete [] pType;
44:     saTaille = rhs.GetTaille();
45:     pType = new int[saTaille];
46:     for (int i = 0; i < saTaille; i++)
47:     {
48:        pType[i] = rhs[i];
49:     }
50:     return *this;
51:  }
52:
53:  Array::Array(const Array &rhs)
54:  {
55:     saTaille = rhs.GetTaille();
56:     pType = new int[saTaille];
57:     for (int i = 0; i < saTaille; i++)
58:     {
59:        pType[i] = rhs[i];
```

```
    }
}

int& Array::operator[](int indice)
{
    int size = GetTaille();
    if (indice >= 0 && indice < GetTaille())
       return pType[indice];
    throw xLimites();
    return pType[0];
}

const int& Array::operator[](int indice) const
{
    int maTaille = GetTaille();
    if (indice >= 0 && indice < GetTaille())
       return pType[indice];
    throw xLimites();
    return pType[0]; // pour MSC
}

ostream& operator<< (ostream& out, const Array& unTableau)
{
    for (int i = 0; i < unTableau.GetTaille(); i++)
       out << "[" << i << "] " << unTableau[i] << endl;
    return out;
}

int main()
{
    Array TabInt(20);
    try
    {
       for (int j = 0; j< 100; j++)
       {
          TabInt [j] = j;
          cout << "TabInt [" << j << "] OK..." << endl;
       }
    }
    catch (Array::xLimites)
    {
       cout << "Impossible de traiter l'entrée !" << endl;
    }
    cout << "Fin." << endl;
    return 0;
}
```

Voici le résultat de l'exécution du programme :

```
TabInt [0] OK...
TabInt [1] OK...
TabInt [2] OK...
TabInt [3] OK...
TabInt [4] OK...
TabInt [5] OK...
TabInt [6] OK...
TabInt [7] OK...
TabInt [8] OK...
TabInt [9] OK...
TabInt [10] OK...
TabInt [11] OK...
TabInt [12] OK...
TabInt [13] OK...
TabInt [14] OK...
TabInt [15] OK...
TabInt [16] OK...
TabInt [17] OK...
TabInt [18] OK...
TabInt [19] OK...
Impossible de traiter l'entrée !
Fin.
```

Le Listing 20.4 présente une classe `Array` très simple mais, cette fois-ci, on a ajouté le traitement des exceptions pour traiter les cas où l'on tente d'accéder en dehors des limites du tableau.

La ligne 24 déclare la nouvelle classe `xLimites` au sein même de la déclaration de `Array`.

Cette classe, qui ne contient aucune donnée, n'est qu'une classe comme les autres, à qui le compilateur attribuera automatiquement un constructeur, un destructeur, un constructeur copie et l'opérateur d'affectation (égal) par défaut. Elle a donc uniquement ces quatre méthodes.

Le fait de déclarer la classe exception dans `Array` n'a d'autre but que de grouper ces deux classes. Comme on l'a expliqué au Chapitre 16, `Array` ne bénéficiera pas pour autant d'un accès particulier à `xLimites`, et celle-ci n'aura pas non plus un accès privilégié aux membres de `Array`.

Les lignes 63-70 et 72-79 modifient les opérateurs d'indexation pour qu'ils testent l'indice demandé. Si sa valeur ne se trouve pas dans la bonne plage de valeurs, ils lancent la classe `xLimites` comme une exception. Les parenthèses sont nécessaires pour différencier l'appel au constructeur de `xLimites` et l'utilisation d'une constante énumérée.

À la ligne 90, la fonction principale déclare un objet `Array` pouvant contenir vingt valeurs. Le bloc `try` se trouve aux lignes 91 à 98 et tente d'ajouter 101 entiers au tableau.

Le gestionnaire destiné à intercepter les exceptions `xLimites` est déclaré à la ligne 99.

Chaque élément du tableau est initialisé dans le bloc `try` qui est créé dans le programme. Lorsque `j` atteint la valeur `20` (à la ligne 93), on accède à l'élément à cet indice, le test de la ligne 66 échoue, et `operateur[]` lance une exception `xLimites` à la ligne 67.

L'exécution du programme se poursuit avec le bloc `catch` de la ligne 99, qui traite l'exception sur la même ligne pour afficher un simple message d'erreur. L'exécution reprend à la fin du bloc `catch`, à la ligne 102.

Placement des blocs *try* et *catch*

Les actions qui peuvent être à l'origine d'exceptions ne sont pas toujours faciles à identifier et il n'est donc pas toujours évident de savoir ou placer les blocs `try`. De la même façon, il faut savoir où traiter les exceptions qui ont été lancées. Vous pourriez décider de lancer toutes les exceptions liées à la mémoire à l'endroit où cette mémoire est allouée, puis de traiter ces exceptions à un niveau plus élevé dans le programme, dans la partie correspondant à l'interface utilisateur, par exemple.

Pour choisir l'emplacement des blocs `try`, recherchez les endroits du programme qui contiennent des allocations de mémoire ou des utilisations de ressources. Recherchez également les endroits susceptibles de provoquer des accès en dehors d'un tableau, les saisies incorrectes, et ainsi de suite. Au pire, placez tout le code de la fonction `main()` dans un bloc `try/catch`, bien qu'il soit généralement situé dans les fonctions de haut niveau, en particulier celles qui connaissent l'interface utilisateur du programme. Une classe utilitaire, par exemple, ne doit généralement pas intercepter les exceptions qui doivent être signalées à l'utilisateur, car elles peuvent être aussi bien employées dans des programmes avec des fenêtres qu'avec des programmes en mode console, voire dans des programmes qui communiquent avec les utilisateurs *via* le Web ou la messagerie.

Fonctionnement de la détection des exceptions

Le principe de fonctionnement des exceptions est le suivant : quand une exception est lancé, la pile des appels est examinée. Cette pile contient la liste des appels de fonction créée dés qu'une partie du programme invoque une autre fonction.

La pile des appels mémorise le chemin d'exécution. Si la fonction `main()` appelle la fonction `Animal::GetNourriture()`, et que `GetNourriture()` appelle `Animal::ChercheNourriture()` qui à son tour appelle `fstream::operator>>()`, tous ces appels seront

stockés dans la pile. Une fonction récursive peut donc apparaître plusieurs fois dans cette pile.

L'exception entraîne le déroulement de la pile pour chaque bloc de code. À mesure que la pile est déroulée, les destructeurs des objets locaux placés sur la pile sont appelés et ces objets sont détruits.

Une ou plusieurs instructions `catch` suivent chaque bloc `try`. Si l'exception correspond à l'une des instructions `catch`, elle est considérée comme traitée par l'exécution de cette instruction. Si elle ne correspond à aucun `catch`, le déroulement de la pile se poursuit.

Si l'exception remonte ainsi jusqu'au début du programme (`main()`) sans avoir été traitée, un gestionnaire intégré est appelé pour interrompre le programme.

Il est important de noter que le déroulement de la pile pour traiter une exception est une opération à sens unique. À mesure qu'elle se produit, la pile est déroulée et les objets qu'elle contient sont détruits. Il n'y a aucune possibilité de retour en arrière. Une fois l'exception traitée, le programme reprend l'exécution après le bloc `try` dans lequel était intégrée l'instruction `catch` qui a traité l'exception.

Dans le Listing 20.4, par exemple, l'exécution va continuer en ligne 101, la première ligne après le bloc `try` de l'instruction `catch` qui a traité l'exception `xLimites`. N'oubliez jamais que lorsqu'une exception est lancée, l'exécution du programme reprend après le bloc `catch`, pas après l'endroit d'où a été déclenché l'exception.

Utilisation de plusieurs clauses *catch*

Une exception peut être provoquée pour plusieurs raisons. On peut alors placer plusieurs instructions `catch` à la suite, un peu comme pour les différents cas d'une instruction `switch`. L'équivalent de la clause `default` est l'instruction `catch(...)` qui permet de traiter tous les types d'exceptions qui n'ont pas été prévus. Le Listing 20.5 illustre les conditions d'exception multiples.

Listing 20.5 : Exceptions multiples

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int TailleDefaut = 10;
4:
5:  class Array
6:  {
7:  public:
8:      // constructeurs
9:      Array(int taille = TailleDefaut);
```

```
10:     Array(const Array &rhs);
11:     ~Array() { delete [] pType;}
12:
13:     // opérateurs
14:     Array& operator=(const Array&);
15:     int& operator[](int indice);
16:     const int& operator[](int indice) const;
17:
18:     // méthodes d'accès
19:     int GetTaille() const { return saTaille; }
20:
21:     // fonction amie
22:     friend ostream& operator<< (ostream&, const Array&);
23:
24:     // définit les classes d'exception
25:     class xLimites {};
26:     class xTropGrand {};
27:     class xTropPetit{};
28:     class xZero {};
29:     class xNegatif {};
30:  private:
31:     int *pType;
32:     int  saTaille;
33:  };
34:
35:  int& Array::operator[](int indice)
36:  {
37:     int taille = GetTaille();
38:     if (indice >= 0 && indice < GetTaille())
39:        return pType[indice];
40:     throw xLimites();
41:     return pType[0];
42:  }
43:
44:
45:  const int& Array::operator[](int indice) const
46:  {
47:     int maTaille = GetTaille();
48:     if (indice >= 0 && indice < GetTaille())
49:        return pType[indice];
50:     throw xLimites();
51:
52:     return pType[0];  // pour MFC
53:  }
```

```
54:
55:
56:  Array::Array(int taille):
57:     saTaille(taille)
58:  {
59:     if (taille == 0)
60:        throw xZero();
61:     if (taille < 10)
62:        throw xTropPetit();
63:     if (taille > 30000)
64:        throw xTropGrand();
65:     if (taille < 1)
66:        throw xNegatif();
67:
68:     pType = new int[taille];
69:     for (int i = 0; i < taille; i++)
70:        pType[i] = 0;
71:  }
72:
73:  int main()
74:  {
75:     try
76:     {
77:        Array TabInt(0);
78:        for (int j = 0; j < 100; j++)
79:        {
80:           TabInt[j] = j;
81:           cout << "TabInt[" << j << "] OK..." << endl;
82:        }
83:     }
84:     catch (Array::xLimites)
85:     {
86:        cout << "Impossible de traiter votre entrée !" << endl;
87:     }
88:     catch (Array::xTropGrand)
89:     {
90:        cout << "Ce tableau est trop grand..." << endl;
91:     }
92:     catch (Array::xTropPetit)
93:     {
94:        cout << "Ce tableau est trop petit..." << endl;
95:     }
96:     catch (Array::xZero)
97:     {
98:        cout << "Vous avez demandé un tableau";
99:        cout << " de zéro objets !" << endl;
```

```
100:     }
101:     catch (...)
102:     {
103:        cout << "Un incident s'est produit !" << endl;
104:     }
105:     cout << "Fin." << endl;
106:     return 0;
107:  }
```

On obtient le résultat suivant :

```
Vous avez demandé un tableau de zéro objets !
Fin.
```

Les lignes 25 à 29 créent quatre nouvelles classes : `xTropGrand`, `xTropPetit`, `xZero`, et `xNegatif`. Le constructeur teste maintenant la taille qui lui a été transmise (lignes 56 à 71) et lance une exception si cette taille est trop petite, trop grande, négative ou nulle.

On a modifié le bloc `try` en y ajoutant des instructions `catch` qui traitent chaque cas, sauf `xNegatif` : cette dernière sera donc prise en charge par dernière instruction `catch(...)`, à la ligne 101.

Testez ce programme en utilisant diverses valeurs pour la taille du tableau. Si vous choisissez une taille égale à `-5`, vous remarquerez que l'exception `xNegatif` n'est pas lancée : ceci est dû à l'ordre des tests dans le constructeur puisque la condition `taille < 10` est évaluée avant `taille < 1`. Il suffit donc d'inverser les lignes 61-62 avec les lignes 65-66 pour corriger ce problème puis de recompiler.



Lorsque le constructeur a été appelé, de la mémoire a été allouée à l'objet. Le lancement d'une exception depuis le constructeur peut donc laisser l'objet alloué, mais inutilisable. Il faut donc généralement entourer le constructeur d'un bloc `try`/`catch` et, en cas d'exception, marquer l'objet (en interne) comme inutilisable. Chaque fonction membre devrait vérifier cette balise "valide" pour être sûre que d'autres erreurs ne surgiront pas lorsque l'on utilisera un objet dont l'initialisation a été interrompue.

Hiérarchies des exceptions

Les exceptions sont des classes et donc peuvent être dérivées. Il peut être intéressant de créer une classe `xTaille`, puis d'en dériver `xZero`, `xTropPetit`, `xTropGrand` et `xNegativf`. Certaines fonctions pourront ainsi se contenter de capturer toutes les erreurs `xTaille`, quelles qu'elles soient, alors que d'autres s'intéresseront à un type spécifique de cette hiérarchie. Le Listing 20.6 illustre cette idée.

Listing 20.6 : Hiérarchies de classes et exceptions

```
 0:  #include <iostream>
 1:  using namespace std;
 2:
 3:  const int TailleDefaut = 10;
 4:
 5:  class Array
 6:  {
 7:  public:
 8:     // constructeurs
 9:     Array(int taille = TailleDefaut);
10:     Array(const Array &rhs);
11:     ~Array() { delete [] pType;}
12:
13:     // opérateurs
14:     Array& operator=(const Array&);
15:     int& operator[](int indice);
16:     const int& operator[](int indice) const;
17:
18:     // méthode d'accès
19:     int GetTaille() const { return saTaille; }
20:
21:     // fonction amie
22:     friend ostream& operator<< (ostream&, const Array&);
23:
24:     // définit les classes d'exception
25:     class xLimites {};
26:     class xTaille {};
27:     class xTropGrand: public xTaille {};
28:     class xTropPetit: public xTaille {};
29:     class xZero  : public xTropPetit {};
30:     class xNegatif  : public xTaille {};
31:  private:
32:     int *pType;
33:     int  saTaille;
34:  };
35:
36:
37:  Array::Array(int taille):
38:     saTaille(taille)
39:  {
40:     if (taille == 0)
41:        throw xZero();
42:     if (taille > 30000)
43:        throw xTropGrand();
44:     if (taille < 1)
```

```
45:        throw xNegatif();
46:     if (taille < 10)
47:        throw xTropPetit();
48:
49:     pType = new int[taille];
50:     for (int i = 0; i < taille; i++)
51:        pType[i] = 0;
52:  }
53:
54:  int& Array::operator[](int indice)
55:  {
56:     int taille = GetTaille();
57:     if (indice >= 0 && indice < taille)
58:        return pType[indice];
59:     throw xLimites();
60:     return pType[0];  // pour MFC
61:  }
62:
63:  const int& Array::operator[](int indice) const
64:  {
65:     int maTaille = GetTaille();
66:
67:     if (indice >= 0 && indice < maTaille)
68:        return pType[indice];
69:     throw xLimites();
70:
71:     return pType[0];  // pour MFC
72:  }
73:
74:  int main()
75:  {
76:     try
77:     {
78:        Array TabInt(0);
79:        for (int j = 0; j < 100; j++)
80:        {
81:           TabInt[j] = j;
82:           cout << "TabInt[" << j << "] OK..." << endl;
83:        }
84:     }
85:     catch (Array::xLimites)
86:     {
87:        cout << "Impossible de traiter votre entrée !" << endl;
88:     }
89:     catch (Array::xTropGrand)
90:     {
91:        cout << "Ce tableau est trop grand..." << endl;
```

```
 92:     }
 93:
 94:     catch (Array::xTropPetit)
 95:     {
 96:        cout << "Ce tableau est trop petit..." << endl;
 97:     }
 98:     catch (Array::xZero)
 99:     {
100:        cout << "Vous avez demandé un tableau";
101:        cout << " de zéro objets !" << endl;
102:     }
103:     catch (...)
104:     {
105:        cout << "Un incident s'est produit !" << endl;
106:     }
107:     cout << "Fin." << endl;
108:     return 0;
109:  }
```

On obtient le résultat suivant :

```
Ce tableau est trop petit...
Fin.
```

Les modifications les plus significatives se situent aux lignes 27 à 30, qui décrivent la hiérarchie des classes. Les classes `xTropGrand`, `xTropPetit` et `xNegatif` dérivent désormais de `xTaille`, et `xZero` dérive de `xTropPetit`.

Vous remarquerez qu'on a créé un tableau de taille zéro, mais que l'on n'a pourtant pas obtenu l'exception attendue ! En examinant les blocs `catch`, vous pouvez constater que l'exception `xTropPetit` est examinée avant l'exception `xZero`. Notre objet `xZero` étant aussi un objet `xTropPetit`, c'est le gestionnaire de cette dernière exception qui traite le problème.

L'ordre des gestionnaires d'exception est important et les plus spécifiques doivent être placés en tête alors que les plus généraux doivent apparaître à la fin. Pour corriger le problème précédent, il suffit donc d'inverser les deux gestionnaires de `xTropPetit` et `xZero`.

Données dans les exceptions et noms des objets exceptions

Connaître le type de l'exception est souvent insuffisant quand on veut traiter correctement les erreurs. Les classes d'exceptions se comportent comme toute autre classe. Vous pouvez

donc leur associer des données et les initialiser dans le constructeur. Celles-ci pourront ensuite être consultées à tout moment. Le Listing 20.7 en présente un exemple.

Listing 20.7 : Récupération de données dans un objet exception

```
 0:  #include <iostream>
 1:  using namespace std;
 2:
 3:  const int TailleDefaut = 10;
 4:
 5:  class Array
 6:  {
 7:  public:
 8:     // constructeurs
 9:     Array(int taille = TailleDefaut);
10:     Array(const Array &rhs);
11:     ~Array() { delete [] pType;}
12:
13:     // opérateurs
14:     Array& operator=(const Array&);
15:     int& operator[](int indice);
16:     const int& operator[](int indice) const;
17:
18:     // méthode d'accès
19:     int GetTaille() const { return saTaille; }
20:
21:     // fonction amie
22:     friend ostream& operator<< (ostream&, const Array&);
23:
24:     // définit les classes d'exception
25:     class xLimites {};
26:     class xTaille
27:     {
28:     public:
29:        xTaille(int taille):saTaille(taille) {}
30:        ~xTaille(){}
31:        int GetTaille() { return saTaille; }
32:     private:
33:        int saTaille;
34:     };
35:
36:     class xTropGrand: public xTaille
37:     {
38:     public:
39:       xTropGrand(int taille):xTaille(taille) {}
40:     };
41:
```

```
42:     class xTropPetit: public xTaille
43:     {
44:     public:
45:       xTropPetit(int taille):xTaille(taille) {}
46:     };
47:
48:     class xZero: public xTropPetit
49:     {
50:     public:
51:        xZero(int taille):xTropPetit(taille) {}
52:     };
53:
54:     class xNegatif : public xTaille
55:     {
56:     public:
57:        xNegatif(int taille):xTaille(taille) {}
58:     };
59:
60:  private:
61:     int *pType;
62:     int  saTaille;
63:  };
64:
65:
66:  Array::Array(int taille):
67:    saTaille(taille)
68:  {
69:     if (taille == 0)
70:        throw xZero(taille);
71:     if (taille > 30000)
72:        throw xTropGrand(taille);
73:     if (taille < 1)
74:        throw xNegatif(taille);
75:     if (taille < 10)
76:        throw xTropPetit(taille);
77:
78:     pType = new int[taille];
79:     for (int i = 0; i < taille; i++)
80:        pType[i] = 0;
81:  }
82:
83:
84:  int& Array::operator[] (int indice)
85:  {
86:     int taille = GetTaille();
87:     if (indice >= 0 && indice < taille)
88:        return pType[indice];
```

```
 89:     throw xLimites();
 90:     return pType[0];
 91:  }
 92:
 93:  const int& Array::operator[] (int indice) const
 94:  {
 95:    int taille = GetTaille();
 96:    if (indice >= 0 && indice < taille)
 97:       return pType[indice];
 98:    throw xLimites();
 99:    return pType[0];
100:  }
101:
102:  int main()
103:  {
104:     try
105:     {
106:        Array TabInt(9);
107:        for (int j = 0; j < 100; j++)
108:        {
109:           TabInt[j] = j;
110:           cout << "TabInt[" << j << "] OK..." << endl;
111:        }
112:     }
113:     catch (Array::xLimites)
114:     {
115:        cout << "Impossible de traiter l'entrée !" << endl;
116:     }
117:     catch (Array::xZero uneException)
118:     {
119:        cout << "Vous avez demandé un tableau de zéro objet !"
119a:            << endl;
120:        cout << "Taille indiquée : "
120a:            << uneException.GetTaille() << endl;
121:     }
122:     catch (Array::xTropGrand uneException)
123:     {
124:        cout << "Ce tableau est trop grand..." << endl;
125:        cout << "Taille indiquée : "
125a:            << uneException.GetTaille() << endl;
126:     }
127:     catch (Array::xTropPetit uneException)
128:     {
129:        cout << "Ce tableau est trop petit..." << endl;
130:        cout << "Taille indiquée : "
130a:            << uneException.GetTaille() << endl;
131:     }
```

```
132:     catch (...)
133:     {
134:        cout << "Un incident inconnu s'est produit !\n";
135:     }
136:     cout << "Fin." << endl;
137:     return 0;
138:  }
```

Ce programme produit le résultat suivant :

```
Ce tableau est trop petit...
Taille indiquée : 9
Fin.
```

La déclaration de la classe `xTaille` a été modifiée pour inclure la variable membre `saTaille` (ligne 33) et la fonction membre `getTaille()` (ligne 31). En outre, on a ajouté en ligne 29 un constructeur prenant un entier en paramètre, afin d'initialiser la variable membre `saTaille`.

Les classes dérivées déclarent un constructeur qui se contente d'initialiser la classe de base. Aucune autre fonction n'a été déclarée afin de réduire la taille de notre exemple.

Les instructions `catch` des lignes 113 à 135 ont été modifiée pour nommer l'exception qu'elles interceptent : `uneException`. Elles utilisent cet objet pour accéder au contenu de `saTaille`.



N'oubliez jamais que si vous construisez une exception, c'est que vous vous trouvez dans une situation exceptionnelle : une erreur s'est produite et votre exception ne doit pas créer le même problème. Si vous créez, par exemple, une exception `OutOfMemory`, vous ne devrez pas allouer de mémoire dans son constructeur.

L'affichage du message approprié par chaque instruction `catch` est fastidieux et peut être à l'origine de nouvelles erreurs. Cette tâche appartient à l'objet qui connaît son type et la valeur qu'il a reçue. Le Listing 20.8 traite le problème avec une approche plus "orientée objet", en utilisant des fonctions virtuelles pour que chaque exception "fasse ce qu'il faut".

Listing 20.8 : Passage par référence et utilisation des fonctions virtuelles dans les exceptions

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int TailleDefaut = 10;
4:
```

```
 5:  class Array
 6:  {
 7:  public:
 8:  // constructeurs
 9:     Array(int taille = TailleDefaut);
10:     Array(const Array &rhs);
11:     ~Array() { delete [] pType;}
12:
13:     // opérateurs
14:     Array& operator=(const Array&);
15:     int& operator[](int indice);
16:     const int& operator[](int indice) const;
17:
18:     // méthode d'accès
19:     int GetTaille() const { return saTaille; }
20:
21:     // fonction amie
22:     friend ostream& operator<<
23:     (ostream&, const Array&);
24:
25:     // définit les classes d'exception
26:     class xLimites {};
27:     class xTaille
28:     {
29:     public:
30:        xTaille(int taille):saTaille(taille) {}
31:        ~xTaille(){}
32:        virtual int GetTaille() { return saTaille; }
33:        virtual void AffErreur()
34:        {
35:           cout << "Erreur de taille. Taille indiquée : ";
36:           cout << saTaille << endl;
37:        }
38:     protected:
39:        int saTaille;
40:     };
41:
42:     class xTropGrand: public xTaille
43:     {
44:     public:
45:        xTropGrand(int taille):xTaille(taille){}
46:        virtual void AffErreur()
47:        {
48:           cout << "Trop grand ! Taille indiquée : ";
49:           cout << xTaille::saTaille << endl;
50:        }
51:     };
```

```
52:
53:     class xTropPetit: public xTaille
54:     {
55:     public:
56:        xTropPetit(int taille):xTaille(taille){}
57:        virtual void AffErreur()
58:        {
59:           cout << "Trop petit ! Taille indiquée : ";
60:           cout << xTaille::saTaille << endl;
61:        }
62:     };
63:
64:     class xZero  : public xTropPetit
65:     {
66:     public:
67:        xZero(int taille):xTropPetit(taille){}
68:        virtual void AffErreur()
69:        {
70:           cout << "Zéro !!. Taille indiquée : " ;
71:           cout << xTaille::saTaille << endl;
72:        }
73:     };
74:
75:     class xNegatif : public xTaille
76:     {
77:     public:
78:        xNegatif(int taille):xTaille(taille){}
79:        virtual void AffErreur()
80:        {
81:           cout << "Négatif ! Taille indiquée : ";
82:           cout << xTaille::saTaille << endl;
83:        }
84:     };
85:
86:  private:
87:     int *pType;
88:     int  saTaille;
89:  };
90:
91:  Array::Array(int taille):
92:     saTaille(taille)
93:  {
94:     if (taille == 0)
95:        throw xZero(taille);
96:     if (taille > 30000)
97:        throw xTropGrand(taille);
98:     if (taille < 0)
```

```
 99:       throw xNegatif(taille);
100:    if (taille < 10)
101:       throw xTropPetit(taille);
102:
103:    pType = new int[taille];
104:    for (int i = 0; i < taille; i++)
105:       pType[i] = 0;
106: }
107:
108: int& Array::operator[] (int indice)
109: {
110:    int taille = GetTaille();
111:    if (indice >= 0 && indice < taille )
112:       return pType[indice];
113:    throw xLimites();
114:    return pType[0];
115: }
116:
117: const int& Array::operator[] (int indice) const
118: {
119:    int taille = GetTaille();
120:    if (indice >= 0 && indice < taille)
121:       return pType[indice];
122:    throw xLimites();
123:    return pType[0];
124: }
125:
126: int main()
127: {
128:    try
129:    {
130:       Array TabInt(9);
131:       for (int j = 0; j < 100; j++)
132:       {
133:          TabInt[j] = j;
134:          cout << "TabInt[" << j << "] OK..." << endl;
135:       }
136:    }
137:    catch (Array::xLimites)
138:    {
139:       cout << "Impossible de traiter l'entrée !" << endl;
140:    }
141:    catch (Array::xTaille& uneException)
142:    {
```

```
143:        uneException.AffErreur();
144:     }
145:     catch (...)
146:     {
147:        cout << "Un incident s'est produit !" << endl;
148:     }
149:     cout << "Fin." << endl;
150:     return 0;
151:  }
```

Ce programme produit le résultat suivant :

```
Trop petit ! Taille indiquée : 9
Fin.
```

La méthode virtuelle `AffErreur()` est déclarée dans la classe `xTaille` (lignes 33 à 37). Elle affiche un message d'erreur et la taille réelle de la classe. Cette fonction est surchargée dans chacune des classes dérivées.

À la ligne 141, l'objet exception est déclaré dans le gestionnaire d'exceptions comme une référence. Lorsque `AffErreur()` est appelée avec la référence d'un objet, c'est la bonne version de cette fonction qui est exécutée grâce au polymorphisme. Le code est plus facile à comprendre et donc à maintenir.

Exceptions et modèles

Lorsque vous créez des exceptions pour fonctionner avec des modèles, vous avez le choix entre créer une exception pour chaque instance du modèle ou utiliser des classes d'exceptions déclarées en dehors de la déclaration du modèle. Le Listing 20.9 illustre les deux solutions.

Listing 20.9 : Utilisation des exceptions avec les modèles

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  const int TailleDefaut = 10;
4:  class xLimites {};
5:
6:  template <class T>
7:  class Array
8:  {
9:  public:
```

```
10:     // constructeurs
11:     Array(int taille = TailleDefaut);
12:     Array(const Array &rhs);
13:     ~Array() { delete [] pType;}
14:
15:     // opérateurs
16:     Array& operator=(const Array<T>&);
17:     T& operator[](int indice);
18:     const T& operator[](int indice) const;
19:
20:     // méthode d'accès
21:     int GetTaille() const { return saTaille; }
22:
23:     // fonction amie
24:     friend ostream& operator<< (ostream&, const Array<T>&);
25:
26:     // définit les classes d'exception
27:
28:     class xTaille {};
29:
30:  private:
31:     int *pType;
32:     int  saTaille;
33:  };
34:
35:  template <class T>
36:  Array<T>::Array(int taille):
37:     saTaille(taille)
38:  {
39:     if (taille < 10 || taille > 30000)
40:        throw xTaille();
41:     pType = new T[taille];
42:     for (int i = 0; i < taille; i++)
43:        pType[i] = 0;
44:  }
45:
46:  template <class T>
47:  Array<T>& Array<T>::operator=(const Array<T> &rhs)
48:  {
49:     if (this == &rhs)
50:        return *this;
51:     delete [] pType;
52:     saTaille = rhs.GetTaille();
53:     pType = new T[saTaille];
54:     for (int i = 0; i < saTaille; i++)
55:        pType[i] = rhs[i];
56:  }
```

```
template <class T>
Array<T>::Array(const Array<T> &rhs)
{
   saTaille = rhs.GetTaille();
   pType = new T[saTaille];
   for (int i = 0; i < saTaille; i++)
      pType[i] = rhs[i];
}

template <class T>
T& Array<T>::operator[](int indice)
{
   int taille = GetTaille();
   if (indice >= 0 && indice < taille)
      return pType[indice];
   throw xLimites();
   return pType[0];
}

template <class T>
const T& Array<T>::operator[](int indice) const
{
   int taille = GetTaille();
   if (indice >= 0 && indice < taille)
      return pType[indice];
   throw xLimites();
}

template <class T>
ostream& operator<< (ostream& out, const Array<T>& unTab)
{
   for (int i = 0; i < unTab.GetTaille(); i++)
      out << "[" << i << "] " << unTab[i] << endl;
   return out;
}


int main()
{
   try
   {
      Array<int> TabInt(9);
      for (int j = 0; j < 100; j++)
      {
         TabInt[j] = j;
         cout << "TabInt[" << j << "] OK..." << endl;
      }
```

```
104:     }
105:     catch (xLimites)
106:     {
107:        cout << "Impossible de traiter l'entrée !" << endl;
108:     }
109:     catch (Array<int>::xTaille)
110:     {
111:        cout << "Taille incorrecte !" << endl;
112:     }
113:
114:     cout << "Fin." << endl;
115:     return 0;
116:  }
```

Ce programme produit le résultat suivant :

```
Taille incorrecte !
Fin.
```

La première exception, `xLimites`, est déclarée à la ligne 4 en dehors de la définition du modèle. La seconde exception, `xTaille`, est déclarée dans la définition du modèle, à la ligne 28.

L'exception `xLimites` n'est donc pas liée à la classe modèle, mais on peut l'utiliser comme toute autre classe. `xTaille`, au contraire, est liée au modèle et doit être appelée sur la base de l'instanciation de `Array`. La syntaxe des deux instructions `catch` est donc différente : la ligne 105 utilise `catch(xLimites)` alors que la ligne 109 utilise `catch(Array<int>::xTaille)`. Cette dernière ligne est liée à l'instanciation d'un `Array` d'entiers.

Exceptions et erreurs

Faut-il utiliser les exceptions pour le traitement des erreurs classiques ? L'avis des programmeurs C++ est partagé. Certains pensent que, par leur nature même, les exceptions devraient être réservées aux circonstances exceptionnelles mais prévisibles (d'où leurs noms !), que le programmeur doit anticiper et qu'elles ne devraient donc pas être servir à traiter les erreurs classiques.

D'autres pensent que les exceptions représentent une méthode efficace pour traverser plusieurs couches d'appels de fonctions sans courir le risque de provoquer des fuites mémoires. Un exemple classique est celui où l'utilisateur demande une action dans un environnement GUI. La partie de code qui récupère la demande doit appeler une fonction membre dans un gestionnaire de dialogue, qui appelle à son tour le code qui va traiter la demande, qui appelle le code qui choisit la boîte de dialogue à utiliser, qui appelle le code

qui affiche la boîte de dialogue, qui appelle enfin le code qui va traiter l'entrée de l'utilisateur. Si l'utilisateur clique sur le bouton Annuler, le code doit revenir à la toute première méthode appelée, où la requête initiale était traitée.

Une solution à ce problème consiste à placer un bloc `try` autour de l'appel initial et à intercepter `AnnulerDialog` comme une exception pouvant être lancée par le gestionnaire du bouton Annuler. Cette méthode est sûre et efficace, mais actionner le bouton Annuler est un comportement classique du programme, pas une circonstance exceptionnelle.

Il existe une façon raisonnable d'aborder la question : cette utilisation des exceptions rend-elle le code plus simple ou plus compliqué ? Y a-t-il ainsi moins ou plus de risques d'erreurs ou de fuites de mémoire ? Le code sera-t-il plus facile ou plus dur à maintenir ? Vous trouverez les réponses en analysant les compromis possibles car il n'existe pas de réponse toute faite.

Pourriture du code

La pourriture du code est un phénomène bien connu de détérioration du logiciel lorsque celui-ci est négligé. Un programme bien conçu, parfaitement testé et corrigé, aura inévitablement besoin d'être entretenu. Après quelques mois, votre client découvrira qu'une moisissure verte recouvre votre code et que la plupart de vos objets commencent à s'écailler.

À part livrer votre code source dans un conteneur sous vide, votre seule protection sera d'écrire du code facile à corriger dès que les problèmes surgiront, car ils ne manqueront pas d'apparaître tôt ou tard.



La pourriture du code est une plaisanterie de programmeur, mais elle enseigne une chose importante. Les programmes sont de plus en plus complexes et les erreurs peuvent rester cachées très longtemps. Protégez-vous en produisant du code facile à maintenir.

Ceci signifie que votre code doit être clair et correctement commenté dans ses parties les plus compliquées. Après six mois, vous le consulterez avec les yeux d'un étranger en vous demandant quel esprit torturé a pu produire une telle chose...

Recherche des erreurs

Presque tous les environnements modernes de développement incluent un ou plusieurs débogueurs. Ceux-ci vous permettent d'exécuter le code de vos programmes, instruction par instruction, en examinant la modification des variables tout au long du traitement.

Tous les compilateurs vous autorisent à compiler avec ou sans symboles. Dans le cas d'une compilation avec symbole, le compilateur doit créer les correspondances nécessaires entre le code source et le programme produit : c'est ce qui permettra au débogueur de pointer sur la ligne de code qui correspond à l'action suivante du programme.

Les débogueurs symboliques plein écran vous facilitent encore plus la tâche puisqu'ils peuvent lire votre code source et l'afficher dans une fenêtre. Vous pouvez ensuite sauter les appels de fonction ou demander au débogueur d'exécuter la fonction, ligne par ligne.

Avec la plupart des débogueurs, vous pouvez passer du code source à la sortie du programme pour examiner le résultat de chaque instruction exécutée. Vous avez même la possibilité de contrôler l'état courant des variables, de visualiser les structures de données complexes et la valeur des données membres dans une classe. Vous pouvez également contrôler les valeurs réelles en mémoire des divers pointeurs et autres emplacements mémoire. Vous pouvez également effectuer plusieurs types de contrôles depuis le débogueur en incluant des points d'arrêt, des points de contrôle, en examinant la mémoire et en visualisant le code assembleur.

Points d'arrêt

Les points de d'arrêt sont des instructions destinées au débogueur qui demandent l'arrêt du programme à une ligne donnée. Ils permettent d'exécuter normalement le programme jusqu'à la ligne indiquée. Les points d'arrêt facilitent donc l'examen de l'état des variables avant et après une ligne de code critique.

Points de contrôle

Vous pouvez demander au débogueur de vous montrer la valeur d'une variable particulière, ou de faire une pause lorsque cette variable est lue ou écrite. Les points de contrôle vous permettent de définir ces conditions et même de modifier la valeur d'une variable pendant l'exécution du programme.

Examen de la mémoire

Il est quelquefois important de pouvoir visualiser les valeurs réelles stockées en mémoire. Les débogueurs modernes peuvent montrer ces valeurs sous la forme de la variable réelle : les chaînes sont affichées avec des caractères, les entiers longs sous la forme de nombres plutôt que quatre octets, et ainsi de suite. Les débogueurs les plus sophistiqués permettent même de visualiser les classes complètes, avec la valeur de toutes les variables membres, y compris le pointeur `this`.

Assembleur

La lecture du code source est généralement suffisante pour dénicher les erreurs, mais si toutes les tentatives ont échoué, vous pouvez demander au débogueur d'afficher le code assembleur produit pour chaque ligne du code source. Vous pouvez ainsi examiner les registres mémoire et les indicateurs (*flags*) pour vous plonger dans le fonctionnement interne du programme.

Apprenez à utiliser votre débogueur car c'est l'outil le plus efficace pour la recherche des erreurs. Les erreurs d'exécution sont les plus difficiles à trouver, mais un débogueur puissant rend possible, sinon facile, leur élimination quasi complète.

Questions-réponses

Q Quel intérêt y a-t-il à provoquer des exceptions ? Pourquoi ne pas traiter les erreurs là où elles se trouvent ?

R La même erreur se retrouve souvent à divers endroits dans le programme. Les exceptions permettent de centraliser la gestion des erreurs. En outre, la partie du code qui produit l'erreur peut ne pas être l'endroit idéal pour savoir comment traiter cette erreur.

Q Pourquoi faut-il produire un objet ? Pourquoi ne pas passer simplement un code de l'erreur ?

R Les objets sont plus souples et plus puissants que les codes d'erreurs. Ils véhiculent davantage d'informations et le mécanisme constructeur/destructeur permet de créer et supprimer les ressources qui peuvent être nécessaires à un traitement correct de l'exception.

Q Pourquoi ne pas utiliser les exceptions pour traiter d'autres conditions que les erreurs ? Il pourrait être intéressant d'avoir la possibilité de revenir très vite à des zones de code précédentes, même s'il n'y a pas eu de condition exceptionnelle.

R Effectivement, certains programmeurs C++ utilisent les exceptions de cette façon. Le danger est que ces exceptions peuvent créer des fuites mémoires lorsque la pile est déroulée et que certains objets peuvent être laissés par inadvertance sur le tas. Une bonne technique de programmation et un bon compilateur permettent généralement d'éviter ces problèmes. C'est donc essentiellement une question de goût, certains programmeurs pensent que les exceptions, par nature, ne sont pas destinées à traiter des conditions normales.

Q Doit-on intercepter une condition là où le bloc `try` l'a créée ?

R Non, l'exception peut être interceptée n'importe où dans la pile des appels. Elle remonte dans cette pile jusqu'à ce qu'elle soit traitée.

Q Pourquoi utiliser un débogueur alors que vous pouvez employer `cout` et d'autres instructions du même genre ?

R Un débogueur fournit un mécanisme beaucoup plus efficace pour parcourir le code pas à pas et visualiser la modification des valeurs, sans avoir besoin de polluer le code avec des milliers d'instructions de débogage. Il existe en outre un risque significatif à chaque fois que vous ajoutez ou supprimez des lignes de code. Si vous venez de régler un problème grâce au débogage et que vous supprimez accidentellement une vraie ligne de code en effaçant la ligne de débogage, la situation ne se sera pas meilleure.

Testez vos connaissances

1. Qu'est-ce qu'une exception ?
2. Qu'est-ce qu'un bloc `try` ?
3. Qu'est-ce qu'une instruction `catch` ?
4. Quel type d'information une exception peut-elle contenir ?
5. Quand les objets exceptions sont-ils créés ?
6. Faut-il transmettre les exceptions par valeur ou par référence ?
7. Une instruction `catch` qui contrôle une classe de base va-t-elle intercepter une exception dérivée ?
8. Si deux instructions `catch` sont utilisées, l'une pour une classe de base, l'autre pour une classe dérivée, laquelle doit intervenir en premier ?
9. Que signifie `catch(...)` ?
10. Qu'est-ce qu'un point d'arrêt ?

Exercices

1. Créez un bloc `try`, une instruction `catch` et une exception simple.
2. Modifiez la réponse précédente pour placer une donnée et une méthode d'accès dans l'exception et utilisez cette méthode dans le bloc `catch`.
3. Transformez la classe de l'Exercice 2 pour créer une hiérarchie d'exceptions. Modifiez le bloc `catch` pour utiliser les objets dérivés et les objets de base.
4. Modifiez le programme de l'Exercice 3 pour obtenir trois niveaux d'appels de fonction.

5. CHERCHEZ L'ERREUR :

```
#include "stringc.h"          // notre classe chaîne

class xOutOfMemory
{
public:
     xOutOfMemory( const String& ou ) : endroit( ou ){}
     ~xOutOfMemory(){}
     virtual String ou(){ return endroit };
private:
     String endroit;
}

int main()
{
     try
     {
          char *var = new char;
          if ( var == 0 )
               throw xOutOfMemory();
     }
     catch( xOutOfMemory& uneException )
     {
          cout << "Mémoire insuffisante à l'adresse "
               << uneException.endroit() << endl;
     }
     return 0;
}
```

Ce listing présente la gestion des exceptions lors de la survenue d'une erreur pour cause de mémoire insuffisante.



21

Et maintenant ?

Au sommaire de ce chapitre

- La compilation conditionnelle et la façon de la gérer
- L'écriture de macros en utilisant le préprocesseur
- L'utilisation du préprocesseur pour rechercher les erreurs
- La manipulation des bits et leur utilisation comme indicateurs
- Les étapes suivantes pour utiliser efficacement le langage C++

Félicitations ! Vous devriez maintenant avoir atteint un bon niveau de programmation en C++. Vous ne devez cependant pas relâcher vos efforts parce que vous aurez toujours quelque chose de nouveau à apprendre. Ce chapitre apporte quelques précisions supplémentaires et donne quelques pistes pour la poursuite de votre étude.

Vos fichiers source sont constitués de code C++. Ce code est interprété par le compilateur pour être transformé en programme exécutable. Cependant, le compilateur n'intervient qu'après le passage du préprocesseur, qui offre donc l'opportunité d'effectuer une compilation conditionnelle.

Le préprocesseur et le compilateur

À chaque fois que vous exécutez le compilateur, le préprocesseur s'exécute d'abord. Celui-ci recherche les instructions qui lui sont destinées, celles qui commencent par le symbole dièse (#). Chacune de ces instructions entraîne une modification du code source. Le résultat obtenu est un nouveau fichier source, un fichier temporaire que vous ne voyez pas en temps normal, mais que vous pouvez sauvegarder par l'intermédiaire du compilateur si vous le désirez.

Le compilateur ne lit pas le fichier source initial : c'est la sortie du préprocesseur qu'il compile. Vous avez déjà constaté cet effet avec la directive `#include`, qui demande au préprocesseur de trouver le fichier indiqué et de placer son contenu dans le fichier temporaire en remplacement de la directive. Cela revient donc exactement à saisir le contenu du fichier inclus dans le code source : lorsque le compilateur verra votre code, le fichier inclus y sera.



Presque tous les compilateurs possèdent une option que vous pouvez activer dans l'environnement de développement intégré (IDE) ou sur la ligne de commande et qui vous permet de demander la sauvegarde du fichier intermédiaire. Consultez la documentation de votre compilateur si vous voulez l'utiliser.

La directive de préprocesseur *#define*

Vous pouvez créer des substitutions de chaînes grâce à la commande `#define`. Si vous écrivez :

```
#define GRAND 512
```

vous demandez au préprocesseur de remplacer toutes les occurrences de `GRAND` par la chaîne `512`. Il ne s'agit pas d'une chaîne au sens C++ de ce terme. Les caractères `512` seront simplement substitués dans le code source à chaque fois que le mot `GRAND` sera rencontré. Si vous écrivez :

```
#define GRAND 512

int monTableau[GRAND];
```

le fichier intermédiaire produit par le préprocesseur sera :

```
int monTableau[512];
```

Vous remarquerez que la directive `#define` a disparu. Les directives du préprocesseur sont toutes supprimées du fichier intermédiaire ; elles n'apparaissent plus dans le code source final.

Utilisation de *#define* pour les constantes

Une des utilisations de `#define` est un substitut aux constantes. Cette technique est cependant souvent déconseillée car `#define` réalise une simple substitution de chaîne, sans aucun contrôle de type. L'utilisation du mot-clé `const` présente de très nombreux avantages par rapport à cette utilisation de `#define`.

Utilisation de *#define* pour les tests

Une autre façon d'utiliser `#define` consiste à déclarer simplement qu'une chaîne de caractères particulière est définie. Exemple :

```
#define DEBUG
```

Vous pourrez tester ensuite si `DEBUG` a été défini et agir en conséquence. N'oubliez pas que tout ceci se passe au niveau du préprocesseur, pas au niveau du compilateur, ni au cours de l'exécution du programme.

Lorsque le préprocesseur lit la directive `#if defined`, il recherche si vous avez défini la valeur qui la suit. Si c'est le cas, `defined` est évaluée à vrai : tout ce qui se trouve entre `#if defined DEBUG` et `#endif` est écrit dans le fichier intermédiaire et sera donc compilé. Dans le cas contraire, la partie entre `#if defined DEBUG` et `#endif` ne sera pas écrite dans le fichier intermédiaire, comme si elle n'était jamais apparue dans le code source.

```
#if defined DEBUG
cout << "Debug est défini";
#endif
```

Vous pouvez aussi tester si une valeur n'est pas définie, en utilisant l'opérateur `not` avec la directive `defined` :

```
#if !defined DEBUG
cout << "Debug n'est pas défini";
#endif
```

Il existe aussi un raccourci pour cela, `#ifndef` :

```
#ifndef DEBUG
cout << "Debug n'est pas défini.";
#endif
```

`#ifndef` est l'inverse logique de `#ifdef`. L'évaluation de cette instruction donne `true` si la chaîne n'a pas été définie dans les lignes précédentes du fichier.

Notez que tous ces tests exigent `#endif` pour indiquer la fin du code concerné.

La commande *#else*

La commande `#else` peut être insérée entre `#ifdef` ou `#ifndef` et le `#endif` de fin. Le Listing 21.1 illustre l'utilisation de ces divers termes.

Listing 21.1 : Utilisation de *#define*

```
 0:  #define DemoVersion
 1:  #define SW_VERSION 5
 2:  #include <iostream>
 3:
 4:  using std::endl;
 5:  using std::cout;
 6:
 7:  int main()
 8:  {
 9:    cout << "Contrôle des définitions de DemoVersion, ";
10:    cout << "SW_VERSION et WINDOWS_VERSION..." << endl;
11:
12:     #ifdef DemoVersion
13:        cout << "DemoVersion définie." << endl;
14:     #else
15:        cout << "DemoVersion non définie." << endl;
16:     #endif
17:
18:     #ifndef SW_VERSION
19:        cout << "SW_VERSION non définie !" << endl;
20:     #else
21:        cout << "SW_VERSION définie avec : "
22:             << SW_VERSION << endl;
23:     #endif
24:
25:     #ifdef WINDOWS_VERSION
26:        cout << "WINDOWS_VERSION définie !" << endl;
27:     #else
28:        cout << "WINDOWS_VERSION n'est pas définie." << endl;
29:     #endif
30:
31:     cout << "Fin." << endl;
32:     return 0;
33:  }
```

Vous obtenez le résultat suivant :

```
Contrôle des définitions de DemoVersion, SW_VERSION et WINDOWS_VERSION...
DemoVersion définie.
SW_VERSION définie avec : 5
WINDOWS_VERSION n'est pas définie.
Fin.
```

`VersionDemo` et `SW_VERSION` sont définies aux lignes 0 et 1, `SW_VERSION` étant associée à la chaîne `5`. La définition de `VersionDemo` est testée à la ligne 12 : comme le résultat de ce test renvoie `true` (bien que cette constante ne soit associée aucune valeur), la chaîne de la ligne 11 est affichée.

La ligne 18 teste si `SW_VERSION` n'est pas défini. Cette constante étant définie, ce test renvoie `false` et l'exécution se poursuit à la ligne 21. La chaîne `5` est alors substituée au mot `SW_VERSION`, c'est la ligne suivante qui sera vue par le compilateur :

```
cout << "SW_VERSION  définie avec : " << 5 << endl;
```

Vous pouvez remarquer que le premier mot `SW_VERSION` n'a pas été remplacé parce qu'il se trouve entre guillemets. Le second a cependant été remplacé et le compilateur voit donc la valeur 5 comme si vous l'aviez ici vous-même placé là.

Pour terminer, le programme teste la présence de la définition de `WINDOWS_VERSION` à la ligne 25. Ce mot n'étant pas défini, le test échoue et le message de la ligne 28 est affiché.

Inclusions et clauses d'inclusion

Vous créerez des projets composés de plusieurs fichiers. Vos répertoires seront probablement organisés de sorte que chaque classe ait son propre fichier en-tête (avec l'extension `.hpp`, par exemple) et son propre fichier d'implémentation (avec l'extension `.cpp`) contenant le code source de ses méthodes.

La fonction `main()` de votre programme sera dans son propre fichier `.cpp` et tous les fichiers `.cpp` seront compilés pour produire des fichiers `.obj` qui seront ensuite pris en charge par l'éditeur de liens pour produire un seul programme exécutable.

Vos programmes utiliseront des méthodes de plusieurs classes et plusieurs fichiers en-têtes seront donc inclus dans chaque fichier. En outre, les fichiers en-têtes ont eux-même souvent besoin d'en inclure d'autres : celui d'une déclaration de classe dérivée doit, par exemple, inclure le fichier en-tête de sa classe de base.

Imaginez que la classe `Animal` soit déclarée dans le fichier `animal.hpp`. La classe `Chien` (qui dérive de la classe `Animal`) doit inclure le fichier `animal.hpp` dans `chien.hpp` pour

hériter de `Animal`. Le fichier en-tête de `Chat` devra également inclure `animal.hpp` pour la même raison.

Si vous créez un programme qui utilise à la fois `Chat` et `Chien`, vous risquez donc d'inclure deux fois `animal.hpp`, ce qui produira une erreur de compilation parce que l'on ne peut pas déclarer deux fois la même classe (`Animal`), même si ces déclarations sont identiques.

Vous pouvez résoudre ce problème avec des clauses d'inclusion. Entourez le contenu du fichier en-tête `animal.hpp` par ces lignes :

```
#ifndef ANIMAL_HPP
#define ANIMAL_HPP
...                     // code du fichier complet
#endif
```

Si le terme `ANIMAL_HPP` n'a pas été défini, les lignes suivantes seront exécutées. Elles définiront donc `ANIMAL.HPP`, puis incluront le fichier entier.

La seconde fois que le programme incluera ce fichier, il lira la première ligne mais, cette fois-ci, le test échouera car vous avez déjà défini `ANIMAL.HPP`. Le préprocesseur ne traitera donc aucune ligne jusqu'à la prochaine directive `#else` (il n'y en pas ici) ou `#endif` (qui se trouve ici à la fin du fichier). La classe ne sera donc jamais définie deux fois.

Le nom de la constante symbolique (`ANIMAL.HPP`) n'a pas d'importance, bien qu'il soit courant d'utiliser le nom du fichier en majuscules en remplaçant le point par un blanc souligné. Il ne s'agit toutefois que d'une convention ; cependant, comme vous ne pouvez pas donner le même nom à deux fichiers différents, cette convention fonctionne bien.



N'hésitez pas à utiliser des clauses d'inclusion car elles peuvent vous faire gagner des heures de débogage.

Les macros

La directive `#define` permet également de créer des *macro-fonctions* (souvent appelées simplement *macros)*. Une macro est un symbole créé avec `#define` et qui reçoit un argument comme une fonction. Le préprocesseur substituera la macro et son paramètre par la chaîne de substitution dans laquelle le paramètre aura lui-même été substitué. Vous pouvez définir, par exemple, la macro `DOUBLE` de la façon suivante :

```
#define DOUBLE(x) ((x) * 2)
```

puis écrire dans votre code :

```
DOUBLE(4)
```

Le préprocesseur remplacera la chaîne `DOUBLE(4)` par la valeur `( (4) * 2 )`, puis remplacera cette expression par `4 * 2`, soit `8`.

Une macro peut avoir plusieurs paramètres, et chaque paramètre peut être utilisé plusieurs fois dans le texte de remplacement. `MAX` et `MIN` sont deux macros courantes :

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

Dans une définition de macro, la première parenthèse de la liste de paramètres doit suivre immédiatement le nom de la macro, sans espace car le préprocesseur, contrairement au compilateur, n'est pas indifférent aux espaces. S'il y a un espace, il effectuera une substitution standard comme nous l'avons vu précédemment.

Si vous écriviez :

```
#define MAX (x,y) ( (x) > (y) ? (x) : (y) )
```

et que vous utilisiez `MAX` de cette façon :

```
int x = 5, y = 7, z;
z = MAX(x,y);
```

le code intermédiaire sera le suivant :

```
int x = 5, y = 7, z;
z = (x,y) ( (x) > (y)  ? (x) : (y) ) (x,y)
```

Il y a donc eu une simple substitution de texte au lieu de l'appel de la macro. Le mot-clé `MAX` a été remplacé par `(x,y) ((x) > (y) ? (x) : (y))`, suivi du `(x,y)` qui lui-même suivait `MAX`.

En supprimant l'espace entre `MAX` et `(x,y)`, le code intermédiaire devient :

```
int x = 5, y = 7, z;
a = ( (5) > (7) ? (5) : (7) );
```

qui, bien sûr, est évalué à `7`.

Pourquoi des parenthèses ?

Vous vous demandez peut-être pourquoi il y a autant de parenthèses dans la plupart des macros présentées jusqu'ici. Le préprocesseur n'impose pas d'encadrer les paramètres de parenthèses dans la chaîne de substitution, mais ces parenthèses évitent certains effets secondaires quand vous transmettez des valeurs compliquées à une macro. Si vous définissez `MAX` de la façon suivante, par exemple :

```
#define MAX(x,y) x > y ? x : y
```

puis que vous transmettiez les valeurs 5 et 7, la macro va fonctionner comme prévu. Si vous transmettez au contraire une expression plus compliquée, vous risquez d'obtenir des résultats inattendus comme illustré dans le Listing 21.2.

Listing 21.2 : Utilisation des parenthèses dans les macros

```
 0:  // Listing 21.2 expansion de macro
 1:  #include <iostream>
 2:  using namespace std;
 3:
 4:  #define CUBE(a) ( (a) * (a) * (a) )
 5:  #define TROIS(a) a * a * a
 6:
 7:  int main()
 8:  {
 9:     long x = 5;
10:     long y = CUBE(x);
11:     long z = TROIS(x);
12:
13:     cout << "y : " << y << endl;
14:     cout << "z : " << z << endl;
15:
16:     long a = 5, b = 7;
17:     y = CUBE(a + b);
18:     z = TROIS(a + b);
19:
20:     cout << "y : " << y << endl;
21:     cout << "z : " << z << endl;
22:     return 0;
23:  }
```

L'exécution du programme donne le résultat suivant :

```
y : 125
z : 125
y : 1728
z : 82
```

La macro `CUBE` est définie à la ligne 4 et le paramètre `x` est indiqué chaque fois entre parenthèses. À la ligne 5, la macro `TROIS` est définie sans parenthèses.

Ces macros sont tout d'abord utilisées aux lignes 10 et 11 en transmettant la valeur 5 comme paramètre et le résultat obtenu est satisfaisant. `CUBE(5)` se développe en `( (5) * (5) * (5) )` qui est évalué en 125 et `TROIS(5)` se développe en `5 * 5 * 5` qui est aussi évalué en 125.

Pour la seconde utilisation, aux lignes 16 à 18, le paramètre transmis est `5+7`. Dans ce cas, l'évaluation de `CUBE(5 + 7)` donne le résultat :

```
( (5+7) * (5+7) * (5+7) )
```

qui est évalué comme :

```
 ( (12) * (12) * (12) )
```

qui donne enfin 1728. `TROIS(5 + 7)` va cependant être évalué comme :

```
5 + 7 * 5 + 7 * 5 + 7
```

La multiplication étant réalisée avant l'addition, cette expression devient :

```
5 +  (7 * 5) +  (7 * 5) +  7
```

qui est évaluée en :

```
5 +  (35) +  (35) +  7
```

qui donne finalement 82. Comme vous pouvez le constater, l'absence des parenthèses a provoqué une erreur.

Manipulation des chaînes

Le préprocesseur possède deux opérateurs pour manipuler des chaînes dans des macros. L'opérateur (#) remplace ce qui le suit par une chaîne entre guillemets, et l'opérateur de concaténation relie deux chaînes en une seule.

L'opérateur

L'opérateur # place entre guillemets les caractères qui le suivent jusqu'au premier espace. Si vous écrivez :

```
#define ECRITCHAINE(x) cout << #x
```

puis que vous appellez :

```
ECRITCHAINE(ceci est une chaîne);
```

Le précompilateur va transformer cette ligne en :

```
cout << "ceci est une chaîne";
```

La concaténation

L'opérateur de concaténation permet d'obtenir un nouveau mot à partir de plusieurs termes. Ce nouveau mot pourra être utilisé pour un nom de classe, un nom de variable, un indice de un tableau et partout où une suite de caractères peut apparaître.

Supposons que vous ayez cinq fonctions nommées `fUneImpression`, `fDeuxImpression`, `fTroisImpression`, `fQuatreImpression` et `fCinqImpression`. Vous pouvez déclarer :

```
#define fIMPRESSION(x) f ## x ## Impression
```

puis utiliser `fIMPRESSION(Deux)` pour produire `fDeuxImpression` ou `fIMPRESSION(Trois)` pour obtenir `fTroisImpression`.

Rappelez-vous que la classe `ListePieces` ne pouvait recevoir que des objets de type `Liste`. Supposons que cette liste fonctionne bien et que vous ayez besoin de constituer des listes d'animaux, de voitures, d'ordinateurs, et ainsi de suite.

Une approche consisterait à créer `ListeAnimal`, `ListeVoiture` `ListeOrdinateur`, etc., et à couper puis coller le code. Ceci deviendrait rapidement un cauchemar, car tout changement dans une liste devra être répercuté dans les autres.

Une solution consiste à utiliser les macros avec l'opérateur de concaténation. Vous pourriez définir par exemple :

```
#define ListeDe(Type)  class Liste##Type \
{ \
public: \
Liste##Type(){} \
private:          \
int saLongueur; \
};
```

Cet exemple est exagérément simple, mais l'idée consiste à placer toutes les méthodes et les données nécessaires dans la définition de la macro. Lorsque vous voudrez créer une `ListeAnimal`, il suffira alors d'écrire :

```
ListeDe(Animal)
```

Cet appel sera transformé en une déclaration de la classe `ListeAnimal`. Cette solution présente cependant quelques inconvénients qui ont été évoqués lorsque nous avons présenté les modèles au Chapitre 19.

Les macros prédéfinies

De nombreux compilateurs prédéfinissent un certain nombre des macros très utiles comme `__DATE__`, `__TIME__`, `__LINE__` et `__FILE__`. Ces noms sont entourés de deux blancs soulignés pour réduire le risque de conflits avec ceux de vos programmes.

Lorsque le préprocesseur rencontre l'une de ces macros, il réalise la substitution appropriée. `__DATE__` est remplacée par la date courante, `__TIME__` par l'heure courante, `__LINE__` et `__FILE__` respectivement par le numéro de ligne du code source et par le nom du fichier. Cette substitution est réalisée pendant la précompilation, pas au moment de l'exécution. Si vous demandez au programme d'afficher `__DATE__`, vous n'obtiendrez pas donc la date en cours, mais la date à laquelle le programme aura été compilé. Ces macros prédéfinies sont très utiles pour le débogage.

La macro *assert()*

Beaucoup de compilateurs reconnaissent la macro `assert()`. Celle-ci renvoie `true` si ses paramètres sont évalués à `true` et réalise une certaine action dans le cas contraire. La plupart des compilateurs interrompent le programme lorsqu'une macro `assert()` échoue, d'autres lancent une exception (voir Chapitre 20).

La macro `assert()` sert au débogage du programme avant sa sortie. En fait, si `DEBUG` n'est pas définie, le préprocesseur ne la transformera pas, de sorte qu'elle ne sera pas incluse dans le source fourni au compilateur. Elle est donc très utile pendant le développement et n'occasionne aucune perte de performance, ni augmentation de la taille de la version exécutable du programme.

Vous pouvez écrire votre propre macro `assert()`. Le Listing 21.3 présente une macro `assert()` personnalisée et montre comment l'utiliser.

Listing 21.3 : Une macro *assert()* simple

```
0:     // Listing 21.3 ASSERTS
1:     #define DEBUG
2:     #include <iostream>
3:  using namespace std;
4:
5:     #ifndef DEBUG
6:        #define ASSERT(x)
7:     #else
8:        #define ASSERT(x) \
9:           if (! (x)) \
```

```
10:          { \
11:            cout << "ERREUR !! Assert " << #x
11a:               << " a échoué" << endl; \
12:            cout << " à la ligne " << __LINE__  << endl; \
13:            cout << " dans le fichier " << __FILE__ << endl;  \
14:          }
15:    #endif
16:
17:  int main()
18:  {
19:     int x = 5;
20:     cout << "Première assertion : " << endl;
21:     ASSERT(x==5);
22:     cout << "\nDeuxième assertion: " << endl;
23:     ASSERT(x != 5);
24:     cout << "\nFin." << endl;
25:     return 0;
26:  }
```

Ce programme produit le résultat suivant :

```
Première assertion:

Deuxième assertion:
ERREUR!! Assert x!=5 a échoué
à la ligne 24
Dans le fichier List2104.cpp
Terminé.
```

`DEBUG` est défini à la ligne 1. En général, on le fait à partir de la ligne de commande (ou de l'IDE) au moment de la compilation, afin d'en contrôler facilement le lancement et l'arrêt. La macro `ASSERT()` est définie aux lignes 8 à 14. Généralement, cette définition est placée dans un fichier en-tête (`assert.hpp`) qui est inclus dans tous les fichiers d'implémentation.

Le terme `DEBUG` est testé à la ligne 5. S'il n'a pas été défini, `ASSERT()` ne produit aucun code. Dans le cas contraire, la fonctionnalité des lignes 8 à 14 est appliquée.

Cette macro est constituée d'une seule longue instruction découpée en sept lignes de code. La valeur transmise sous la forme d'un paramètre est testée à la ligne 9. Si le résultat de son évaluation est `false`, les instructions des lignes 11 à 13 affichent un message d'erreur. Si cette évaluation donne le résultat `true`, aucune action n'est entreprise.

Effets de bord

Il est fréquent de voir apparaître une erreur après avoir enlevé la macro `assert()`. Cela est presque toujours dû aux effets de bord, sur le programme, des actions de `assert()` ou de tout code réservé à la correction des erreurs. Si vous écrivez, par exemple :

```
ASSERT (x = 5)
```

alors que vous vouliez tester `x == 5`, vous créerez une erreur particulièrement difficile à repérer.

Supposons que juste avant cette `assert()`, une fonction ait initialisé `x` à zéro. Dans cette macro `assert()`, vous pensiez tester l'égalité de `x` avec 5 mais, en fait, vous l'avez initialisée à 5. Le test renvoie donc la valeur 5 et, comme cette valeur est différente de zéro, le test est évalué comme vrai.

Après l'exécution de l'instruction `assert()`, `x` est effectivement égale à 5 et le programme se déroule normalement. Vous décidez alors de supprimer `DEBUG`, `assert()` disparaît, `x` conserve la valeur zéro et le programme ne fonctionne plus. Si vous introduisez de nouveau `DEBUG`, l'erreur, bien sûr, disparaîtra. Vous devez donc être très prudent avec le code destiné à la recherche des erreurs, notamment aux effets de bord qu'il peut provoquer.

Les invariants de classe

La plupart des classes possèdent des conditions qui seront toujours vraies après l'appel d'une méthode. Ces *invariants* de classe sont indispensables. Un cercle, par exemple, n'aura jamais un rayon égal à zéro, et l'âge de votre animal sera toujours supérieur à zéro et inférieur à 100.

Il peut être utile de déclarer une méthode `Invariants()` qui ne renvoie la valeur `true` que si chacune de ces conditions est toujours vraie. Vous pourrez ainsi placer `ASSERT(Invariants())` au début et à la fin de chaque méthode de classe. Les deux exceptions sont que l'on ne s'attend pas à ce que `Invariants()` renvoie vrai avant l'exécution du constructeur, ni après celle du destructeur. Le Listing 21.4 illustre l'utilisation d'une méthode `Invariants()` dans une classe très simple.

Listing 21.4 : Utilisation de *Invariants()*

```
0:  #define DEBUG
1:  #define SHOW_INVARIANTS
2:  #include <iostream>
3:  #include <string.h>
4:  using namespace std;
5:
```

```
 6:  #ifndef DEBUG
 7:     #define ASSERT(x)
 8:  #else
 9:     #define ASSERT(x) \
10:         if (! (x)) \
11:         { \
12:            cout << "ERREUR !! Assert " << #x << " a échoué" \
12a:                << endl; \
13:            cout << " à la ligne " << __LINE__  << endl; \
14:            cout << " dans le fichier " << __FILE__ << endl;  \
15:         }
16:  #endif
17:
18:
19:  const int FALSE = 0;
20:  const int TRUE = 1;
21:  typedef int BOOL;
22:
23:
24:  class String
25:  {
26:  public:
27:     // constructeurs
28:     String();
29:     String(const char *const);
30:     String(const String &);
31:     ~String();
32:
33:     char & operator[](int indice);
34:     char operator[](int indice) const;
35:
36:     String & operator= (const String &);
37:     int GetLongueur()const { return saLongueur; }
38:     const char * GetString() const { return saString; }
39:     BOOL Invariants() const;
40:
41:  private:
42:     String (int);         // constructeur privé
43:     char * saString;
44:     // unsigned short saLongueur;
45:     int saLongueur;
46:  };
47:
48:  // Constructeur par défaut qui crée des chaînes de 0 octet
49:  String::String()
50:  {
51:     saString = new char[1];
```

```
52:     saString[0] = '\0';
53:     saLongueur = 0;
54:     ASSERT(Invariants());
55:  }
56:
57:  // constructeur privé, utilisé uniquement par les
58:  // méthodes de la classe pour créer une nouvelle chaîne
59:  // de la taille demandée remplie par des caractères null.
60:  String::String(int longueur)
61:  {
62:     saString = new char[longueur+1];
63:     for (int i = 0; i <= longueur; i++)
64:        saString[i] = '\0';
65:     saLongueur = longueur;
66:     ASSERT(Invariants());
67:  }
68:
69:  // Convertit un tableau de caractères en chaîne
70:  String::String(const char * const chaineC)
71:  {
72:     saLongueur = strlen(chaineC);
73:     saString = new char[saLongueur+1];
74:     for (int i = 0; i < saLongueur; i++)
75:        saString[i] = chaineC[i];
76:     saString[saLongueur]='\0';
77:     ASSERT(Invariants());
78:  }
79:
80:  // constructeur de copie
81:  String::String (const String & rhs)
82:  {
83:     saLongueur = rhs.GetLongueur();
84:     saString = new char[saLongueur+1];
85:     for (int i = 0; i < saLongueur;i++)
86:        saString[i] = rhs[i];
87:     saString[saLongueur] = '\0';
88:     ASSERT(Invariants());
89:  }
90:
91:  // destructeur, libère la mémoire allouée
92:  String::~String ()
93:  {
94:     ASSERT(Invariants());
95:     delete [] saString;
96:     saLongueur = 0;
97:  }
98:
99:  // opérateur d'affectation, libère la mémoire puis
```

```
100:  // copie la chaîne et la taille
101:  String& String::operator=(const String & rhs)
102:  {
103:     ASSERT(Invariants());
104:     if (this == &rhs)
105:        return *this;
106:     delete [] saString;
107:     saLongueur = rhs.GetLongueur();
108:     saString = new char[saLongueur+1];
109:     for (int i = 0; i < saLongueur;i++)
110:        saString[i] = rhs[i];
111:     saString[saLongueur] = '\0';
112:     ASSERT(Invariants());
113:     return *this;
114:  }
115:
116:  //opérateur d'indexation non constant
117:  char & String::operator[](int indice)
118:  {
119:     ASSERT(Invariants());
120:     if (indice > saLongueur)
121:     {
122:        ASSERT(Invariants());
123:        return saString[saLongueur-1];
124:     }
125:     else
126:     {
127:        ASSERT(Invariants());
128:        return saString[indice];
129:     }
130:  }
131:
132:  // opérateur d'indexation constant
133:  char String::operator[](int indice) const
134:  {
135:     ASSERT(Invariants());
136:     char result;
137:     if (indice > saLongueur)
138:        result = saString[saLongueur-1];
139:     else
140:        result = saString[indice];
141:     ASSERT(Invariants());
142:     return result;
143:  }
144:
145:  BOOL String::Invariants() const
146:  {
```

```
147:     #ifdef SHOW_INVARIANTS
148:        cout << "Chaîne testée OK ";
149:     #endif
150:     return ( (saLongueur && saString) ||
150a:              (!saLongueur && !saString) );
151:  }
152:
153:  class Animal
154:  {
155:  public:
156:     Animal():sonAge(1) ,sonNom("John Q. Animal")
157:        {ASSERT(Invariants());}
158:     Animal(int, const String&);
159:     ~Animal(){}
160:     int GetAge() {  ASSERT(Invariants()); return sonAge;}
161:     void SetAge(int Age)
162:     {
163:        ASSERT(Invariants());
164:        sonAge = Age;
165:        ASSERT(Invariants());
166:     }
167:     String& GetNom()
168:     {
169:        ASSERT(Invariants());
170:        return sonNom;
171:     }
172:     void SetNom(const String& nom)
173:     {
174:        ASSERT(Invariants());
175:        sonNom = nom;
176:        ASSERT(Invariants());
177:     }
178:     BOOL Invariants();
179:  private:
180:     int sonAge;
181:     String sonNom;
182:  };
183:
184:  Animal::Animal(int age, const String& nom):
185:     sonAge(age),
186:     sonNom(nom)
187:  {
188:     ASSERT(Invariants());
189:  }
190:
191:  BOOL Animal::Invariants()
192:  {
```

```
193:     #ifdef SHOW_INVARIANTS
194:        cout << "Animal testé OK ";
195:     #endif
196:        return (sonAge > 0 && sonNom.GetLongueur());
197:  }
198:
199:  int main()
200:  {
201:     Animal sparky(5, "Sparky");
202:     cout << endl << sparky.GetNom().GetString() << " a ";
203:     cout << sparky.GetAge() << " ans.";
204:     sparky.SetAge(8);
205:     cout << endl << sparky.GetNom().GetString() << " a ";
206:     cout << sparky.GetAge() << " ans.";
207:     return 0;
208:  }
```

Ce programme produit le résultat suivant :

```
Chaîne testée OK Chaîne testée OK Chaîne testée OK Chaîne testée OK Chaîne
testée OK Chaîne testée OK Chaîne testée OK Chaîne testée OK Chaîne testée OK
Chaîne testée OK Chaîne testée OK Chaîne testée OK Chaîne testée OK Chaîne
testée OK Animal testé OK Chaîne testée OK Animal testé OK
Sparky a Animal testé OK 5 ans.Animal testé OK Animal testé OK Animal testé OK
Sparky a Animal testé OK 8 ans.Chaîne testée OK
```

La macro `ASSERT()` est définie en lignes 9 à 15. Si `DEBUG` a été défini, un message d'erreur est affiché si la macro est évaluée fausse.

La fonction membre `Invariants()` est déclarée à la ligne 39 et définie aux lignes 143 à 150. Le constructeur est déclaré aux lignes 49 à 55 : `Invariants()` n'est appelée qu'après la construction complète de l'objet, ligne 54, pour confirmer que la construction est correcte.

Ce schéma est répété dans le cas des autres constructeurs ; le destructeur, au contraire, n'appelle `Invariants()` que lorsqu'il est sur le point de détruire l'objet. Les fonctions restantes appellent `Invariants()` avant toute action et avant de renvoyer leur résultat. Cela affirme et valide un principe fondamental de C++ : les fonctions membres qui ne sont ni des constructeurs ni des destructeurs doivent travailler avec des objets valides et les laisser dans un état correct.

À la ligne 176, la classe `Animal` déclare sa propre méthode `Invariants()`, qui est implémentée aux lignes 189 à 195. Vous pouvez noter que les fonctions inline, aux lignes 155, 158, 161 et 163, peuvent appeler la méthode `invariants()`.

Affichage des valeurs intermédiaires

La macro `ASSERT()` permet d'affirmer que quelque chose est vrai ou faux, mais elle permet également d'afficher la valeur courante de pointeurs, de variables ou de chaînes de caractères. Cette possibilité est très utile pour contrôler le déroulement du programme et localiser les erreurs de boucles. Le Listing 21.5 illustre cette technique.

Listing 21.5 : Affichage de valeurs en mode *DEBUG*

```
 0:  // Listing 21.5 - Affichage des valeurs en mode DEBUG
 1:  #include <iostream>
 2:  using namespace std;
 3:  #define DEBUG
 4:
 5:  #ifndef DEBUG
 6:     #define PRINT(x)
 7:  #else
 8:     #define PRINT(x) \
 9:          cout << #x << ":\t" << x << endl;
10:  #endif
11:
12:  enum BOOL { FALSE, TRUE } ;
13:
14:  int main()
15:  {
16:     int x = 5;
17:     long y = 738981;
18:     PRINT(x);
19:     for (int i = 0; i < x; i++)
20:     {
21:        PRINT(i);
22:     }
23:
24:     PRINT (y);
25:     PRINT("Hi.");
26:     int *px = &x;
27:     PRINT(px);
28:     PRINT (*px);
29:     return 0;
30:  }
```

Ce programme produit le résultat suivant :

```
x:            5
i:            0
i:            1
```

```
i:              2
i:              3
i:              4
y:              73898
"Hi.":          Hi.
px:             0012FEDC
*px:            5
```

La macro `PRINT()` des lignes 8 et 9 affiche la valeur courante de son paramètre. À la ligne 9, vous pouvez remarquer que l'on commence par transmettre à `cout` le paramètre sous forme de chaîne de caractères (à l'aide de `#`). Si vous avez passé `x`, `cout` affichera donc "x".

`cout` reçoit ensuite la chaîne `" :\t"` afin d'afficher deux points et se déplacer d'une tabulation, puis la valeur du paramètre `(x)` et, enfin, `endl`, qui écrit un retour à la ligne et vide la mémoire tampon.

Sur votre machine, la valeur du pointeur `px` (`0012FEDC`, ici) sera probablement différente.

Macros *vs.* fonctions et modèles

Les macros de C++ présentent quatre inconvénients. Le premier est qu'elles peuvent devenir confuses si elles sont trop longues, car une macro doit être définie sur une seule ligne. Bien que vous puissiez étendre cette ligne en utilisant le caractère (`\`), les macros longues deviennent rapidement très difficiles à gérer.

Le deuxième inconvénient est que ces macros sont développées en ligne à chaque fois qu'elles sont utilisées. Cela signifie que si une macro est appelée une douzaine de fois, la substitution de cette macro apparaîtra douze fois dans le programme au lieu d'une seule fois, comme pour les appels de fonctions. Par contre, et pour cette même raison, les macros sont plus rapides que les appels de fonctions puisqu'elles permettent justement d'éviter le coût de leur appel.

Ce développement des macros en ligne conduit au troisième inconvénient : une macro n'apparaît pas dans le code source intermédiaire qui est utilisé par le compilateur ; elle n'est donc pas prise en compte par la plupart des programmes de correction des erreurs, ce qui complique beaucoup son débogage.

Le dernier inconvénient est le plus grave : les macros n'effectuent aucun contrôle de type. Bien qu'il soit très pratique de pouvoir utiliser n'importe quel type de paramètre dans une macro, cela sape complètement le contrôle des types de C++. La bonne méthode pour éviter ce problème consiste évidemment à utiliser les modèles, qui ont été présentés au Chapitre 19.

Les fonctions *inline*

Il est souvent possible de déclarer une fonction `inline` à la place d'une macro. Le Listing 21.6, par exemple, crée une fonction `cube()` qui se comporte comme la macro `CUBE` du Listing 21.2, mais qui est bien plus sûre car elle bénéficie du contrôle des types.

Listing 21.6 : Utilisation d'une fonction inline au lieu d'une macro

```
 0:  #include <iostream>
 1:  using namespace std;
 2:
 3:     inline unsigned long Carre(unsigned long a) { return a*a; }
 4:     inline unsigned long Cube(unsigned long a)
 5:         { return a * a * a; }
 6:     int main()
 7:     {
 8:        unsigned long x = 1 ;
 9:        for (;;)
10:       {
11:          cout << "Entrez un nombre (0 pour sortir) : ";
12:          cin >> x;
13:          if (x == 0)
14:             break;
15:          cout << "Vous avez tapé : " << x;
16:          cout << ".  Carré(" << x << ") : ";
17:          cout  << Carre(x);
18:          cout<< ". Cube(" << x << ") : ";
19:          cout << Cube(x) << "." << endl;
20:       }
21:       return 0;
22:    }
```

Ce programme produit le résultat suivant :

```
Entrez un nombre (0 pour sortir) : 1
Vous avez tapé : 1. Carré(1): 1. Cube(1): 1.
Entrez un nombre (0 pour sortir): 2
Vous avez tapé: 2. Carré(2) : 4. Cube(2) : 8.
Entrez un nombre (0 pour sortir) : 3
Vous avez tapé: 3. Carré(3) : 9. Cube(3) : 27.
Entrez un nombre (0 pour sortir) : 4
Vous avez tapé: 4. Carré(4) : 16. Cube(4) : 64.
Entrez un nombre (0 pour sortir) : 5
Vous avez tapé: 5. Carré(5) : 25. Cube(5) : 125.
Entrez un nombre (0 pour sortir) : 6
Vous avez tapé: 6. Carré(6) : 36. Cube(6) : 216.
Entrez un nombre (0 pour sortir) : 0
```

Les lignes 3 et 4 déclarent deux fonctions, `Carre()` et `Cube()`. Comme elles sont déclarées inline, elles seront donc développées en ligne à chaque appel, comme les macro. Il n'y aura par conséquent pas de surcharge système induite par un appel de fonction.

"Développée en ligne" signifie que le contenu de la fonction est inséré dans le code pour chaque appel de la fonction (à la ligne 17, par exemple). Il n'y a donc jamais d'appel de fonction, il n'est donc pas nécessaire d'enregistrer l'adresse de retour et les paramètres dans la pile.

La fonction `Carre` est appelée à la ligne 17 et la fonction `Cube` à la ligne 19. Ces deux fonctions étant inline, cela revient exactement à avoir écrit :

```
16:          cout << ".  Carré(" << x << ") : " ;
17:          cout << x * x ;
18:          cout << ". Cube(" << x << ") : " ;
19:          cout << x * x * x << "." << endl;
```

Faire	*Ne pas faire*
● Écrire les noms de macro en CAPITALES. Cette convention est utilisée par tous les programmeurs.	● Créer des effets de bord avec vos macros. N'incrémentez pas de variables et ne leur affectez aucune valeur dans une macro.
● Placer tous les paramètres des macros entre parenthèses dans le corps des macros.	● Utiliser des valeurs `#define` alors qu'une variable constante fonctionnera.

Opérations sur les bits

Il est souvent nécessaire de définir des indicateurs pour suivre l'état d'un objet (Y a-t-il une erreur ? La valeur est-elle vraie ou fausse ? Ce module a-t-il été traité ?).

Les types booléens sont très utiles, mais certaines applications – notamment celles qui utilisent des pilotes de périphériques – exigent d'utiliser les bits individuels d'une variable en guise d'indicateurs.

Chaque octet contient 8 bits. Dans une variable longue, il y a 4 octets, soit 32 possibilités d'indicateurs. Un bit est dit "activé" s'il est à 1 ou "désactivé" s'il est égal à 0. Bien qu'il soit possible d'activer et désactiver les différents bits qui composent un entier long en modifiant sa valeur, cette opération serait fastidieuse et source de confusion.

Les opérateurs bit à bit de C++ permettent de manipuler individuellement les bits d'une variable. Comme ils ressemblent beaucoup aux opérateurs logiques, les programmeurs débutants les confondent souvent, alors que leurs effets sont différents. Ces opérateurs sont présentés dans le Tableau 21.1.

Tableau 21.1 : Opérateurs bit à bit

Symbole	*Opérateur*
&	ET
|	OU
^	OU exclusif
~	Complément à 1

Opérateur ET

L'opérateur ET s'exprime sous la forme d'une esperluette (`&`). Il ne faut pas le confondre avec le ET logique qui, lui, est représenté par les deux signes "`&&`". Le résultat est 1 si les deux bits sont égaux à 1, mais 0 si au moins l'un des deux est égal à 0. Cela se traduit par : "Si le premier bit est à 1 et que le deuxième est aussi à 1, le résultat est 1 ; sinon le résultat est 0."

Opérateur OU

L'opérateur OU est une barre verticale (`|`). Il ne faut pas non plus le confondre avec l'opérateur OU logique qui est représenté par deux barres verticales (`||`). Le résultat est 1 si l'un et/ou l'autre bit est égal à 1. Si aucun bit n'est à 1, la valeur est 0.

Opérateur OU exclusif

L'opérateur OU exclusif est représenté par un accent circonflexe (`^`). Le résultat est 1 si les deux bits sont différents. Le résultat est égal à 0 si les deux bits sont identiques (les deux sont à 1 ou aucun ne l'est).

Opérateur de complément à 1

Cet opérateur représenté par un *tilde* (~) "inverse" chaque bit en mettant à 0 un bit à 1 ou à 1 un bit à 0. Si le nombre vaut `1010 0011`, son complément à 1 est donc `0101 1100`.

Activer des bits

Pour activer ou désactiver un bit particulier, on utilise un *masque* (une suite de 0 et de 1). Pour activer un bit, un utilise ce masque avec un OU ; pour le désactiver, on utilise un ET. Si l'on dispose, par exemple, d'un indicateur de 4 octets et que l'on souhaite activer le bit numéro 8, il suffit de faire un OU de cet indicateur avec le masque 128.

En effet, 128 correspond à la valeur binaire `1000 0000`, c'est-à-dire que seul son huitième bit est à 1. Quelle que soit la valeur de ce bit (1 ou 0) dans l'indicateur, un OU avec la valeur 128 activera donc ce bit sans modifier les autres. Supposons, par exemple, que la valeur des 16 bits soit égale à `1010 0110 0010 0110`. Un OU de cette valeur avec 128 donnera donc :

```
            9 8765 4321
    1010 0110 0010 0110   // bit 8 est désactivé
|   0000 0000 1000 0000   // 128
  ----------------------
    1010 0110 1010 0110   // bit 8 est activé
```

Comme d'habitude, les bits sont comptés de droite à gauche. Les bits de la valeur 128 sont tous nuls sauf le bit 8 (celui que l'on veut activer dans l'indicateur), qui est égal à 1. On remarque également que le nombre de départ `1010 0110 0010 0110` n'est pas modifié par l'opération OU, sauf son huitième bit, qui est passé à 1. S'il avait déjà été à 1, il le serait resté, ce qui est exactement ce que l'on souhaitait obtenir.

Désactiver des bits

Pour désactiver le bit 8 (le mettre à zéro), il suffit d'utiliser l'opérateur ET avec le complément de 128 (c'est-à-dire `0111 1111`). Le nombre d'origine restera inchangé, sauf que son bit 8 sera forcé à 0.

```
  1010 0110 1010 0110  // le bit 8 est activé
& 1111 1111 0111 1111  // ~128
----------------------
  1010 0110 0010 0110  // le bit 8 est désactivé
```

Pour comprendre le résultat, prenez un crayon et un papier. À chaque fois que les deux bits sont à 1, écrivez 1 comme réponse ; si l'un des deux est à 0, inscrivez 0. Comparez le résultat à l'original, vous constaterez qu'ils sont identiques, à part le bit 8 qui a été forcé à 0.

Inverser des bits

Pour inverser le bit 8 — peu importe qu'il soit actif ou inactif —, utilisez l'opérateur OU exclusif avec un masque de 128. Si vous le faites deux fois, vous retrouvez la valeur initiale :

```
  1010 0110 1010 0110  // nombre
^ 0000 0000 1000 0000  // 128
----------------------
  1010 0110 0010 0110  // inversion !
^ 0000 0000 1000 0000  // 128
----------------------
  1010 0110 1010 0110  // retour à l'original
```

Faire	*Ne pas faire*
● Utiliser un masque et l'opérateur OU pour activer des bits. ● Utiliser un masque et l'opérateur ET pour désactiver des bits. ● Utiliser un masque et l'opérateur OU Exclusif pour inverser des bits.	● Confondre les différents opérateurs de bits. ● Oublier les bits à gauche de ceux que vous permutez. Un octet vaut huit bits ; il faut connaître le nombre d'octets de la variable utilisée.

Champs de bits

Dans certains ca, le moindre octet compte et économiser 6 ou 8 octets dans une classe peut faire toute la différence. Si votre classe ou votre structure contient des variables booléennes – ou des variables ayant un choix limité de valeurs – vous pouvez gagner de la place en utilisant les champs de bits.

Parmi les types de données standard du C++, c'est le type `char` qui est le plus petit, puisque sa taille n'est que d'un octet. Vous utiliserez le plus souvent les entiers (`int`) qui ont le plus souvent une taille de 4 octets sur un processeur 32 bits. En utilisant les champs de bits, vous pouvez stocker huit bits dans un type `char`, et trente-deux dans un entier de 4 octets.

Le fonctionnement des champs de bits est le suivant : ils portent un nom et on y accède comme à n'importe quel membre d'une classe. Leur type est toujours déclaré comme `unsigned int`. Après le nom du champ de bits, vous devez placer un caractère deux-points suivi d'un nombre.

Ce nombre indique au compilateur le nombre de bits qui seront assignés à la variable. Si vous écrivez 1, le bit contiendra la valeur 0 ou 1. Si vous écrivez 2, deux bits pourront représenter des nombres : le champ pourra alors représenter 0, 1, 2 ou 3, soit un total de

quatre valeurs. Selon le même principe, un champ de 3 bits pourra représenter huit valeurs, et ainsi de suite. Le Listing 21.7 illustre l'utilisation des champs de bits.

Listing 21.7 : Champs de bits

```
 0:  #include <iostream>
 1:  using namespace std;
 2:  #include <string.h>
 3:
 4:  enum STATUT { TempsPlein, TempsPartiel } ;
 5:  enum GRADE { Etudiant, Diplome } ;
 6:  enum LOGEMENT { ChambreUniversitaire, Exterieur };
 7:  enum REGIME { DemiPension, Pension, WeekEnds, Externe };
 8:
 9:  class etudiant
10:  {
11:  public:
12:     etudiant():
13:        monStatut(TempsPlein),
14:        monGrade(Etudiant),
15:        monLogement(ChambreUniversitaire),
16:        monRegime(Externe)
17:     {}
18:     ~etudiant(){}
19:     STATUT GetStatut();
20:     void SetStatut(STATUT);
21:     unsigned GetRegime() { return monRegime; }
22:
23:  private:
24:     unsigned monStatut : 1;
25:     unsigned monGrade : 1;
26:     unsigned monLogement : 1;
27:     unsigned monRegime : 2;
28:  };
29:
30:  STATUT etudiant::GetStatut()
31:  {
32:     if (monStatut)
33:        return TempsPlein;
34:     else
35:        return TempsPartiel;
36:  }
37:
38:  void etudiant::SetStatut(STATUT statut)
39:  {
40:     monStatut = statut;
41:  }
```

```
42:
43:  int main()
44:  {
45:     etudiant Jean;
46:
47:     if (Jean.GetStatut()== TempsPartiel)
48:        cout << "Jean est à temps partiel" << endl;
49:     else
50:        cout << "Jean est à temps complet" << endl;
51:
52:     Jean.SetStatut(TempsPartiel);
53:
54:     if (Jean.GetStatut())
55:        cout << "Jean est à temps partiel" << endl;
56:     else
57:        cout << "Jean est à temps complet" << endl;
58:
59:     cout << "Jean est " ;
60:
61:     char Regime[80];
62:     switch (Jean.GetRegime())
63:     {
64:     case DemiPension : strcpy(Regime, "Demi-pensionnaire");
64a:                        break;
65:     case Pension: strcpy(Regime, "Pensionnaire"); break;
66:     case WeekEnds: strcpy(Regime, "Pensionnaire le week-end");
66a:                    break;
67:     case Externe: strcpy(Regime, "Externe");break;
68:     default : cout << "Il y a eu un problème !" << endl;
69:               break;
70:     }
71:     cout << Regime << endl;
72:     return 0;
73:  }
```

L'exécution du programme donne le résultat suivant :

```
Jean est à temps partiel
Jean est à temps complet
Jean est Externe.
```

Les lignes 4 à 7 définissent plusieurs types énumérés qui permettent de décrire les valeurs disponibles pour les champs de bits de la classe `etudiant`.

La classe `etudiant` est déclarée aux lignes 9 à 28. Cette classe très simple permet de regrouper toutes les données dans 5 bits aux lignes 24 à 27. Le premier, ligne 24, représente le statut de l'étudiant, temps complet ou temps partiel. Le deuxième, ligne 25, indique si

celui-ci est diplômé ou non. Le troisième, ligne 25, signale si l'étudiant dort à l'internat. Les deux derniers bits représentent les quatre possibilités de pension/repas.

Les méthodes sont créées de la même façon que pour toutes les classes. Le fait que l'on travaille avec des champs de bits plutôt qu'avec des types entiers ou énumérés n'a aucune répercussion.

Bien que cette opération ne soit pas indispensable, la fonction membre `GetStatut()`, aux lignes 30 à 36, analyse le bit booléen et renvoie un type énuméré. Elle aurait pu aussi également renvoyer directement la valeur du champ de bit. Dans ce cas, la conversion aurait été réalisée par le compilateur.

Vous pouvez le vérifier en remplaçant l'implémentation de `GetStatut()` par le code suivant :

```
STATUT student::GetStatut()
{
return monStatut;
}
```

Ces lignes ne devraient modifier en rien le fonctionnement du programme. C'est une simple question de clarté du code.

Notez que le code de la ligne 47 doit contrôler le statut puis afficher le message approprié. Si vous aviez écrit :

```
cout <<"Jean est " << Jean.GetStatut() << endl;
```

vous auriez obtenu l'affichage :

```
Jean est 0
```

Le compilateur ne peut en effet savoir comment convertir la constante énumérée `Temps-Partiel` en texte clair.

Le programme sélectionne ensuite la formule de pension choisie à la ligne 62 et enregistre le message correspondant dans le tampon, qui est ensuite affiché à la ligne 71. Ici aussi, vous auriez pu écrire autrement l'instruction `switch` :

```
case  0: strcpy(Regime, "Demi-pensionnaire"); break;
case  1: strcpy(Regime, "Pensionnaire"); break;
case  2: strcpy(Regime, "Pensionnaire le week-end"); break;
case  3: strcpy(Regime, "Externe");break;
```

Le plus important, dans les champs de bits, est que le client de la classe n'a pas besoin de se soucier de l'implémentation des données. Les champs de bits étant privés, vous pouvez les modifier à tout moment sans toucher à l'interface.

Styles de programmation

Nous avons déjà mentionné l'importance d'adopter un style de programmation cohérent, quel qu'il soit. Un style cohérent faciliter la relecture et la compréhension du code et évite de devoir se rappeler si le nom d'une fonction commence par une majuscule ou une minuscule, par exemple.

Les conseils de cette section ne sont fournis qu'à titre indicatif. Ils sont tirés des directives de projet sur lesquels nous avons travaillé par le passé. Vous pouvez bien sûr créer vos propres règles et ces informations pourraient alors constituer un point de départ.

L'indentation

Si vous utilisez l'indentation, elle ne devrait être que de trois espaces. Assurez-vous que votre éditeur de texte convertisse chaque tabulation en trois espaces.

Les accolades

Les accolades représentent un autre domaine de la programmation assez controversé. Voici quelques suggestions :

- Alignez verticalement les accolades correspondantes.
- Le premier niveau d'accolades dans une définition ou une déclaration doit être placé sur la marge de gauche. Les instructions qu'elles encadrent seront alors indentées. Les autres paires d'accolades seront alignées avec les instructions qu'elles suivent.
- Placez les accolades sur leur propre ligne, comme dans l'exemple suivant :

```
if (condition == true)
{
   j = k;
   UneFonction();
}
m++;
```



L'alignement des accolades peut être controversé. De nombreux programmeurs C++ considèrent que l'accolade ouvrante doit être placée sur la même ligne que la commande à laquelle elle est associée, et que l'accolade fermante s'aligne avec la commande :

```
if (condition == true) {
   j = k;
   UneFonction();
}
```

Toutefois, certains considèrent que ce format est plus difficile à lire car les accolades ne sont pas alignées.

Les lignes longues et la longueur des fonctions

Les lignes doivent pouvoir apparaître dans leur totalité sur l'écran. Le défilement horizontal est une pratique pénible et l'on risque d'oublier le code qui disparaît à droite de l'écran.

Lorsque vous coupez une ligne, indentez les suivantes. Découpez la ligne de façon logique et conservez l'opérateur à la fin de la ligne précédente (et non au début de la suivante). Vous indiquerez ainsi clairement que la ligne n'est pas terminée.

En C++, les fonctions sont généralement beaucoup plus courtes qu'elles ne l'étaient en langage C. Toutefois, l'ancien conseil reste d'actualité : essayez d'obtenir des fonctions qui tiennent sur sur une seule page.

Structurer les instructions *switch*

Pour préserver l'espace horizontal, décalez ces instructions de la manière suivante :

```
switch(variable)
{
   case ValeurUn:
      ActionUne();
      break;
   case ValeurDeux:
      ActionDeux();
      break;
   default:
      assert("Mauvaise action");
      break;
}
```

Comme vous pouvez le constater, les instructions `case` sont légèrement indentées et alignées. De même, les instructions de chaque cas sont alignées. Grâce à cette disposition, on repère facilement une instruction `case` et le code est facile à suivre.

Le texte du programme

Plusieurs astuces permettent d'obtenir un code lisible et d'en simplifier ainsi la maintenance.

- Aérez le texte pour le rendre plus lisible.
- N'utilisez pas d'espaces entre les objets et les tableaux et leurs opérateurs (`.`, `->`, `[]`).
- N'introduisez pas d'espace entre les opérateurs unaires et leurs opérandes. Les opérateurs unaires sont `~`, `++`, `--`, `-`, `*` (pour les pointeurs), `&` (adressage) et `sizeof`.

- Insérez un espace de chaque côté des opérateurs binaires : `+`, `=`, `*`, `/`, `%`, `>>`, `<<`, `<`, `>`, `==`, `!=`, `&`, `|`, `&&`, `||`, `?:`, `=`, `+=`, etc.
- N'ôtez pas les espaces pour signaler l'ordre d'exécution des opérations (`4+ 3*2`).
- Insérez un espace après les virgules et les points-virgules, pas avant.
- Ne mettez pas d'espaces autour des parenthèses.
- Les mots-clés (comme `if`) devraient être mis en valeur à l'aide d'un espace : `if (a == b)`.
- Les commentaires d'une seule ligne ressortent mieux s'ils sont séparés de `//` par un espace.
- Accolez l'indicateur de pointeur ou de référence au nom du type plutôt qu'au nom de la variable :

```
char* ptr;
int&  unInt;
```

plutôt que :

```
char *ptr;
int &unInt;
```

- Ne déclarez pas plusieurs variables sur la même ligne.

Nommer les identificateurs

Voici quelques conseils pour choisir les noms des identificateurs :

- Leur nom doit être assez long pour être descriptif.
- Évitez les abréviations qui ne représentent pas avec précision le mot complet.
- Prenez le temps d'expliquer clairement les choses.
- N'utilisez pas la notation hongroise. Le contrôle des types en C++ est assez efficace et il n'y a donc aucune raison pour l'indiquer dans le nom de la variable. Cette notation perd d'ailleurs de sa signification dans le cas des types utilisateur (classes). Vous pouvez faire une exception en employant un préfixe pour les pointeurs (`p`) et les références (`r`), ainsi que pour les variables membres de classe (`son`, `sa`).
- N'employez des noms courts (`i`, `p`, `x`, etc.) que dans les cas où leur simplicité améliore la lisibilité du code ou lorsque leur utilisation est assez évidente pour que l'on puisse se passer d'un nom descriptif. En général, il est toutefois préférable de l'éviter. Évitez aussi d'utiliser les lettres i, l et o, car elles se confondent facilement avec des chiffres.

- La longueur du nom d'une variable doit être proportionnelle à sa portée.
- Les identificateurs doivent pouvoir être facilement différenciés pour éviter toute confusion.
- Les noms de fonctions (ou de méthodes) sont généralement constitués de verbes ou de groupes verbaux : `Recherche()`, `Initialiser()`, `TrouverParagraphe()`, `AfficherCurseur()`. Les noms de variables sont plutôt constitués de noms abstraits, accompagnés éventuellement d'un autre substantif : `compteur`, `etat`, `hauteurFenetre`, `vitesseVent`. Les variables booléennes doivent être particulièrement descriptives : `fenetreReduite`, `fichierOuvert`.

L'orthographe et la casse

L'orthographe et l'utilisation de la casse ne doivent jamais être négligées.

- Utilisez uniquement des majuscules et séparez les mots des noms `#defined` par le caractère souligné (exemple `MODELE_FICHIER_SOURCE`). Notez cependant que cette notation est rarement utilisée en C++. Utilisez plutôt des constantes et des modèles.
- Pour tous les autres identificateurs, mélangez les majuscules et les minuscules sans introduire de caractères soulignés. Vous pouvez débuter tous les noms de fonctions, méthodes, classes, `typedef` et `struct` par une lettre majuscule. Les éléments comme les données membres ou les variables locales devront plutôt commencer par une minuscule.
- Vous pouvez préfixer les constantes énumérées par des lettres minuscules représentant l'abréviation de l'énumération concernée. Exemple :

```
enum StyleTexte
{
    stNormal,
    stGras,
    stItaliques,
    stSouligne,
};
```

Commentaires

Lors de la saisie d'un programme, vous savez avec précision ce que vous faites. Mais ouvrez de nouveau le fichier source un mois plus tard et certains passages du listing risquent de vous paraître difficiles à déchiffrer. Pour éviter ce désagrément et permettre aux autres de comprendre vos fichiers, insérez des commentaires. Ceux-ci ont pour but d'informer le lecteur sur telle ou telle action du programme. Voici quelques conseils :

- À chaque fois que cela est possible, préférez les commentaires d'une seule ligne (`//`) à ceux sur plusieurs lignes (`/* */`). Réservez plutôt ces derniers aux cas où vous devez commenter des blocs de code qui pourraient contenir des commentaires sur une seule ligne.
- Les commentaires destinés à décrire la structure du programme doivent apporter une valeur ajoutée et non reprendre les détails du code.

  ```
  n++; // on incrémente n de un
  ```

  Ce commentaire n'apporte rien. Décrivez plutôt la sémantique ou les objectifs des fonctions et des blocs de code. Signalez les effets de bord, les types des paramètres et les valeurs renvoyées. Décrivez les hypothèses émises, comme "`suppose que n n'est pas négatif`" ou "`renvoie -1 si x est invalide`". Lorsque la logique du programme se complique, insérez des commentaires en divers points stratégiques du code.
- Faites des phrases complètes. Introduisez la ponctuation et les majuscules de façon appropriée. N'utilisez pas d'abréviations ou de phrases dont le sens n'est pas absolument clair. Ce qui vous semble limpide au moment où vous écrivez le code pourrait devenir confus quelques mois plus tard.
- Insérez des lignes blanches pour aider le lecteur à comprendre le déroulement du programme. Regroupez logiquement les instructions.

Définir les accès

L'accès aux diverses parties du programme doit être aussi cohérent que possible. Voici quelques conseils :

- Codez toujours explictement les indicateurs `public:`, `private:`, `protected:`. N'utilisez pas les valeurs par défaut.
- Fournissez d'abord la liste des membres publics, puis celle des membres protégés et enfin celle des membres privés. Les données membres devront apparaître dans un groupe après les méthodes.
- Placez le constructeur suivi du destructeur dans une section à part. Dans la mesure du possible, regroupez les méthodes surchargées de même nom et faites de même pour les méthodes d'accès.
- Classez par ordre alphabétique les méthodes dans chaque groupe et faites de même pour les variables membres et les noms des fichiers inclus.
- Même si le mot-clé `virtual` est facultatif lors de la substitution, ne l'omettez pas. Vous obtiendrez ainsi une déclaration plus cohérente et vous n'oublierez pas qu'il s'agit d'un élément virtuel.

Les définitions de classes

Lors de la définition des méthodes, essayez de suivre l'ordre des déclarations. La recherche en sera simplifiée.

Lorsque vous définissez une fonction, placez le type du résultat et tous les autres modificateur sur la ligne précédente, de sorte que le noms de la classe et de la fonction débutent dans la marge de gauche. Vous retrouverez ainsi plus facilement les différentes fonctions.

Les fichiers *include*

Essayez au maximum de réduire l'utilisation de `#include` et donc le nombre de fichiers inclus dans les en-têtes. L'idéal serait de se contenter du fichier en-tête de la classe dont on dérive. Les autres fichiers qui seront obligatoirement inclus correspondront aux objets membres de la classe déclarée.

Ne supprimez pas un fichier `include` d'un en-tête uniquement parce que vous supposez que le fichier .cpp l'incluera également. Et n'en ajoutez pas pour essayer d'"aider" d'autres fichiers inclus.



Tous les fichiers en-têtes devraient utiliser les directives `#ifndef...#endif` pour éviter les inclusions multiples.

assert()

Vous avez déjà vu la macro `assert()`. Utilisez-la librement. Cette macro permet de retrouver les erreurs et représente une bonne indication des hypothèses sur lesquelles repose le code. Elle permet d'indiquer de façon très claire ce qui est valide ou ce qui ne l'est pas.

Rendre des éléments constants avec *const*

N'hésitez pas à utiliser `const` pour les paramètres, les variables et les méthodes. Il n'est pas rare d'avoir besoin des deux versions `const` et `non-const` pour une méthode : ce n'est pas une excuse pour en omettre une. Soyez très prudent lorsque vous effectuez un transtypage explicite de `const` vers `non-const` et *vice versa* (parfois, vous n'aurez pas d'autre alternative). Assurez-vous que cette opération est vraiment indispensable et introduisez un commentaire.

Comment poursuivre votre développement C++

Ce livre vous a permis d'acquérir les bases pour devenir un programmeur C++ compétent. Toutefois, il vous reste plusieurs aspects à découvrir. Internet et, en particulier, le World

Wide Web regorgent d'informations de tous types. Vous pouvez rechercher des groupes de discussion ou des sites web avec le mot-clé C++, le nom de votre système d'exploitation, ou `programming` pour obtenir une liste exhaustive.

La plupart des magazines de programmation possèdent maintenant leurs sites web à partir desquels vous pouvez télécharger des articles, dialoguer avec les auteurs, etc. Voici les deux groupes de discussion que nous utilisons régulièrement et que nous vous recommandons : **comp.lang.c++** et **comp.lang.c++.moderated** (leur version francophone est **fr.comp.lang.c++).**

Il existe également des sites comme **http://www.CodeGuru.com** et **http://www.CodeProject.com** où se retrouvent des centaines de milliers de développeurs chaque mois. Ils proposent des articles, des didacticiels, des informations et des discussions sur C++. Il existe également bien d'autres communautés équivalentes.

La plate-forme .NET de Microsoft représente une révolution dans notre façon de programmer pour Internet. Le nouveau langage C# est un élément clé de cette plate-forme.

C# est une extension naturelle du langage C++ ; il permet aux programmeurs C++ d'aborder plus facilement la plate-forme .NET (*via* les extensions gérées de C++).

C# présente toutefois quelques différences avec C++ : l'héritage multiple, par exemple, n'est pas autorisé en C#, même si les interfaces proposent des fonctionnalités similaires. En outre, C# évite les pointeurs, ce qui élimine certains problèmes comme les pointeurs fous, mais qui l'empêche de créer des programmes de très bas niveau ou en temps réel. Enfin, C# utilise un *runtime* et un ramasse-miettes qui libère les ressources lorsqu'elles ne sont plus nécessaires, ce qui épargne au programmeur de devoir le faire.

Le C++ géré provient aussi de Microsoft et fait partie de .NET. En termes simples, il s'agit d'une extension de C++ qui permet d'utiliser toutes les fonctions de .NET, y compris le ramasse-miettes.

Garder le contact

Je serais heureux de recevoir vos commentaires ou vos questions à propos de ce livre et d'y répondre. La meilleure façon de me joindre est de consulter mon site web : **www.libertyassociates.com**.

Faire	*Ne pas faire*
● Consulter d'autres livres. Il y a beaucoup à apprendre et un seul livre ne peut pas vous apporter toutes les informations que vous devez connaître. ● Rejoindre un bon groupe d'utilisateurs C++.	● Lire simplement du code ! La meilleure façon d'apprendre le langage C++ est d'écrire des programmes C++.

Questions-réponses

Q Si C++ offre de meilleures solutions de rechange à l'utilisation du préprocesseur, pourquoi cette option est-elle toujours disponible ?

R La première raison est que C++ est compatible avec le langage C. Toute partie significative de C doit donc être supportée par C++. La seconde est que certaines caractéristiques du préprocesseur sont encore fréquemment utilisées en C++, comme `#ifndef` et `#endif`.

Q Pourquoi utiliser une macro quand l'on peut utiliser une fonction normale ?

R Les macros sont développées en ligne et permettent de ne pas devoir réécrire une commande plusieurs fois avec des changements mineurs. Les modèles offrent généralement une meilleure solution.

Q Comment savoir s'il faut utiliser une macro ou une fonction inline ?

R La plupart du temps, cela n'a pas d'importance. Les macros permettent la substitution de caractères ou la concaténation, qui ne sont pas disponibles avec les fonctions.

Q Quelle est l'alternative au préprocesseur pour afficher des valeurs intermédiaires, pendant la phase de correction des erreurs ?

R La meilleure solution consiste à utiliser des instructions `watch` dans un débogueur ? Pour plus d'informations sur ces instructions, consultez la documentation de votre débogueur ou de votre compilateur.

Q Comment choisir entre l'utilisation de `assert()` et le lancement d'une exception ?

R Si la situation testée peut être vraie même si le programme ne contient pas d'erreur, utilisez une exception. Si la seule raison pour que cette situation soit vraie est la présence d'une erreur dans le programme, utilisez `assert()`.

Q Quand doit-on utiliser les structures de bits plutôt que de simples entiers ?

R Quand la taille de l'objet est capitale. Lorsque vous travaillez avec une mémoire limitée ou avec des logiciels de communication, les types d'enregistrements offerts par ces structures sont essentiels au succès du produit.

Q Puis-je affecter un pointeur à un champ de bit ?

R Non. Les adresses mémoire pointent généralement vers le début d'un octet. Un champ de bits pourrait se trouver au milieu d'un octet.

Q Pourquoi les problèmes de style font-ils couler autant d'encre ?

R Les programmeurs sont très attachés à leurs habitudes. Si vous êtes habitués à l'indentation suivante :

```
if (UneCondition){
     // instructions
}    // accolade de fin
```

vous aurez du mal à changer votre notation. Les nouveaux styles sont peu séduisants et peuvent introduire certaines confusions. Si vous le désirez, vous pouvez vous connecter sur un service en ligne connu et demander l'indentation qui fonctionne le mieux, l'éditeur qui est le mieux adapté au C++ ou le meilleur éditeur de texte. Vous risquez alors de recevoir 10 000 messages, tous en contradiction les uns avec les autres.

Q Avons-nous épuisé ce sujet ?

R Il y a encore dix ans, il était possible de tout connaître des ordinateurs ou, du moins, d'être familier avec tout ce qui les concernait. Cela n'est plus possible aujourd'hui. La technique évolue trop rapidement. Documentez-vous et tenez-vous toujours au courant des dernières modifications en consultant régulièrement les magazines et les services en ligne.

Testez vos connaissances

1. À quoi sert `#ifndef` ?
2. Comment pouvez-vous demander au compilateur d'afficher le fichier intermédiaire pour voir les effets du préprocesseur ?
3. Quelle différence y a-t-il entre `#define debug 0` et `#undef debug` ?
4. Étudiez la macro suivante :

   ```
   #define MOITIE(x) x / 2
   ```

   Quel sera le résultat si elle est appelée avec 4 ?
5. Quel sera le résultat si elle est appelée avec 10+10 ?
6. Comment pourriez-vous modifier la macro `MOITIE` pour éviter d'obtenir des résultats erronés ?

7. Combien de valeurs de bits pourraient être stockées dans une variable de 2 octets ?
8. Combien peut-on stocker de valeurs dans 5 bits ?
9. Quel est le résultat de 0011 1100 | 1111 1111 ?
10. Quel est le résultat de 0011 1100 & 1111 1111 ?

Exercices

1. Écrivez les instructions de contrôle des inclusions multiples pour le fichier en-tête STRING.H.
2. Écrivez une macro `assert()` qui affiche un message d'erreur avec le fichier et le numéro de ligne si le niveau de debug est 2, qui affiche seulement un message si ce niveau est égal à 1 et qui ne fait rien si ce niveau est égal à 0.
3. Écrivez la macro `DPrint`, qui teste si `DEBUG` est définie et qui, si c'est le cas, affiche la valeur transmise en paramètre.
4. Écrivez la déclaration pour stocker le mois, le jour et l'année dans une même variable entière non signée.



A

Mots-clés C++

Les mots-clés sont des mots réservés par le compilateur pour le langage de programmation. Vous ne pouvez pas les utiliser comme noms de classes, de variables ou de fonctions.

```
asm
auto
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
operator
private
protected
public
register
```

```
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while
```

En outre, les mots suivants sont également réservés :

```
and
and_eq
bitand
bitor
compl
not
not_eq
or
or_eq
xor
xor_eq
```



B

Priorité des opérateurs

Il est important de savoir que les opérateurs ont une priorité, mais il n'est pas essentiel de la connaître par cœur.

La priorité est l'ordre dans lequel un programme réalise les opérations dans une formule. Si un opérateur a la priorité sur un autre, il est évalué en premier.

Les opérateurs dont la priorité est plus élevée seront donc évalués avant ceux dont la priorité est moins élevée. Dans le Tableau B.1, plus le rang est petit, plus la priorité est élevée.

Tableau B.1 : Priorité des opérateurs

Rang	*Nom*	*Opérateur*
1	Résolution de portée	`::`
2	Sélection de membre, indication d'indice, appels de fonction, incrémentation et décrémentation suffixe	`. ->` `( )` `++ --`

Tableau B.1 : Priorité des opérateurs *(suite)*

Rang	*Nom*	*Opérateur*
3	`sizeof`, incrémentation et décrémentation préfixe, complément, opérateurs et, non, moins et plus unaires, adresse de et indirection, `new`, `new[]`, `delete`, `delete[]`, transtypage, `sizeof()`,	`++ --` `^ !` `- +` `& *` `()`
4	Sélection de membre pour pointeur	`.* ->*`
5	Multiplication, division, modulo	`* / %`
6	Addition, soustraction	`+ -`
7	Décalage (gauche ou droit)	`<< >>`
8	Inégalité	`< <= > >=`
9	Égalité, différence	`== !=`
10	ET bit à bit	`&`
11	OU exclusif bit à bit	`^`
12	OU bit à bit	`|`
13	ET logique	`&&`
14	OU logique	`||`
15	Conditionnel	`?:`
16	Opérateurs d'affectation	`= *= /= %=` `+= -= <<= >>=` `&= |= ^=`
17	Virgule	



C

Solutions des exercices

Chapitre 1

Testez vos connaissances

1. Les interpréteurs lisent le code source et traduisent directement les instructions du programme en actions, alors que les compilateurs transforment le code source en un programme exécutable ultérieurement.
2. Chaque compilateur est différent. Consultez la documentation qui l'accompagne.
3. La fonction de l'éditeur de liens consiste à lier votre code compilé avec les bibliothèques ou autres sources fournies avec le compilateur. L'éditeur de liens vous permet de construire votre programme par "morceaux", puis de les lier pour obtenir un seul programme plus important.
4. Modification du code source, compilation, édition de liens, test (exécution), puis répétition du processus si nécessaire.

Exercices

1. Ce programme initialise deux variables entières (nombres) puis en affiche la somme, 12, et le produit, 35.
2. Consultez la documentation de votre compilateur.
3. Vous devez placer un symbole # avant le mot `include` de la première ligne.
4. Ce programme affiche le mot `Bonjour` sur la console, suivi d'une nouvelle ligne (retour chariot).

Chapitre 2

Testez vos connaissances

1. À chaque fois que vous lancez le compilateur, le préprocesseur s'exécute d'abord. Il lit votre code source et inclut les fichiers demandés ou toute autre tâche qui lui a été signifiée. Le compilateur s'exécute ensuite pour transformer ce code source pré-traité en code objet.
2. La fonction `main()` est spéciale car elle est appelée automatiquement à chaque fois que votre programme s'exécute. Sa présence est obligatoire dans un programme.
3. Les commentaires C++, sur une seule ligne, commencent avec deux barres obliques (`//`) et se poursuivent jusqu'à la fin de la ligne. Les commentaires de style C, sur plusieurs lignes, sont délimités par la paire `/* ... */`, et tout ce qui se trouve entre ces deux signes (éventuellement sur plusieurs lignes) fait partie du commentaire.
4. Les commentaires de style C++, sur une ligne, peuvent être imbriqués dans un commentaire de style C, sur plusieurs lignes :

```
/* Cette marque initie un commentaire. Tout est ignoré
// y compris ce commentaire sur une ligne,
jusqu'à la marque de fin  */
```

   Vous pouvez, en fait, imbriquer un commentaire multi-lignes dans un commentaire de style C++, mais n'oubliez pas que les commentaires de style C++ se terminent à la fin de la ligne.

5. Les commentaires de style C peuvent occuper plusieurs lignes. Si vous voulez poursuivre les commentaires C++ d'une ligne sur la ligne suivante, vous devez la commencer par deux caractères slash (`//`).

Exercices

1. Voici une réponse possible :

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:    cout << "Le C++, j'adore\n";
6: return 0;
7: }
```

2. Le programme suivant contient une fonction `main()` qui ne fait rien. C'est toutefois un programme complet, qui peut être compilé, lié et exécuté. L'exécution ne produira rien :

```
int main(){}
```

3. Il manque, ligne 4, le premier guillemet de la chaîne à afficher.

4. Voici le programme corrigé :

```
1: #include <iostream.h>
2: main()
3: {
4:    cout << " Y a-t-il un bug dans la salle ?";
5: }
```

   Ce programme affiche :

```
Y a-t-il un bug dans la salle ?
```

5. Voici l'une des solutions possibles :

```
 1:  #include <iostream>
 2:  int Ajouter (int premier, int second)
 3:  {
 4:      -std::cout << "Dans Ajouter(), est reçu " << premier <<
4a:                       " et " <<  second << "\n";
 5:      return (premier + second);
 6:  }
 7:
 8:  int Soustraire (int premier, int second)
 9:  {
10:      -std::cout << "Dans Soustraire(), est recu " <<
10a:                    premier << "  et " << second << "\n";
11:      return (premier - second);
12:  }
```

```
13:
14:  int main()
15:  {
16:      using std::cout;
17:      using std::cin;
18:
19:      cout << "Je suis dans main() !\n";
20:      int a, b, c;
21:      cout << "Entrez deux nombres : ";
22:      cin >> a;
23:      cin >> b;
24:
25:      cout << "\nAppel de Ajouter()\n";
26:      c = Ajouter(a, b);
27:      cout << "\nRetour dans main().\n";
28:      cout << "c a été initialisé à " << c;
29:
30:      cout << "\n\nAppel de Soustraire()\n";
31:      c = Soustraire(a, b);
32:      cout << "\nRetour dans main().\n";
33:      cout << "c a été initialisé à " << c;
34:
35:      cout << "\nActuel...\n\n";
36:      return 0;
37:  }
```

Chapitre 3

Testez vos connaissances

1. Les variables `int` sont des nombres entiers. Les variables en virgule flottante sont des "réels" et ont un point décimal "flottant". Les nombres en virgule flottante peuvent être représentés à l'aide d'une mantisse et d'un exposant.
2. Le mot-clé `unsigned` signifie que l'entier ne stocke que des nombres positifs. Sur la plupart des ordinateurs avec processeur 32 bits, les entiers courts occupent 2 octets et les entiers longs 4. La seule garantie, cependant, est qu'un entier long sera au moins aussi grand ou plus grand qu'un entier ordinaire, qui est lui-même au moins aussi grand qu'un entier court. Généralement, un entier long est deux fois plus grand qu'un entier court.
3. Par définition, le nom d'une constante symbolique doit refléter son rôle dans le programme. Elles peuvent également être redéfinies à un seul endroit dans le code source, le programmeur n'est donc pas obligé de vérifier la totalité du code pour modifier chaque occurrence de la variable.

4. Les variables `const` ont un type bien défini, ce qui permet au compilateur de contrôler la façon dont elles sont implémentées. En outre, le débogueur peut les traiter puisqu'elles existent encore après le passage du préprocesseur. Surtout, l'utilisation de `#define` pour déclarer des constantes n'est plus prise en charge par la norme C++.
5. Un nom de variable bien choisi indique ce que cette variable représente. Un mauvais nom ne contient aucune information. `monAge` et `gensDansLeBus` sont des noms corrects, alors que `x`, `xjk` et `prnd1` ne sont pas explicites.
6. `BLEU` est associé à la valeur 102.
7. Voici les réponses :
   a. Bon
   b. Incorrect
   c. Correct, mais ce n'est pas un bon choix
   d. Bon
   e. Correct, mais ce n'est pas un bon choix

Exercices

1. Voici les types de variables correctes :
   a. Entier court non signé.
   b. Entier long non signé ou flottant non signé.
   c. Double non signé.
   d. Entier court non signé.
2. Voici quelques exemples de noms corrects pour ces variables :
   a. `monAge`
   b. `AireCour`
   c. `EtoilesDansGalaxie`
   d. `MoyPluieJanv`
3. Une déclaration pour PI : `const float PI = 3.14159;`
4. Voici une déclaration et une initialisation de variable :

```
float valPi = PI;
```

Chapitre 4

Testez vos connaissances

1. Une expression est une instruction qui renvoie une valeur.
2. C'est une expression dont la valeur est 12.
3. 50.
4. 1.
5. `Age: 41, a: 39, b: 41.`
6. 14.
7. `if (x = 3)` affecte la valeur 3 et renvoie la valeur 3, qui est considérée comme vraie. `if (x == 3)` teste si `x` est égal à 3 ; elle renvoie vrai si la valeur de `x` est égale à 3 et faux dans le cas contraire.
8. a. est évalué à faux, b. vrai, c. vrai, d. faux et e. vrai.

Exercices

1. Voici une réponse possible :

```
if (x > y)
   x = y;
else        // x > y || y == x
   y = x;
```

2. Voir Exercice 3.
3. Entrer 20, 10, 50 donne pour résultat : `a: 20`, `b: 30` et `c: 10` :

   La ligne 14 contient une affectation et non un test d'égalité.
4. Voir Exercice 5.
5. La ligne 6 attribue la valeur de `a-b` à `c`, la valeur de l'affectation est donc `a(2)` moins `b(2)`, soit 0. Zéro étant évalué comme faux, l'instruction `if` échoue et rien n'est affiché.

Chapitre 5

Testez vos connaissances

1. Le prototype de fonction déclare la fonction, la définition la définit. Le prototype se termine par un point-virgule, pas la définition. À la différence de la définition, la

déclaration peut contenir le mot-clé `inline` et des valeurs de paramètre par défaut. Dans une déclaration, il n'est pas utile d'inclure les noms des paramètres.

2. Non. Les paramètres sont identifiés par leur position, jamais par leur nom.
3. Vous déclarez la fonction avec un type renvoyé `void`.
4. En l'absence de type de résultat, une fonction renvoie par défaut une valeur entière. Il est conseillé de prendre la bonne habitude de toujours déclarer le type du résultat.
5. Une variable locale est une variable qui est passée ou déclarée dans un bloc, en général une fonction. Elle n'est accessible que dans ce bloc d'instructions.
6. La portée se rapporte à la visibilité et à la durée de vie des variables globales et locales. Elle est généralement délimitée par une accolade ouvrante et une accolade fermante.
7. La récursivité est la possibilité, pour une fonction, de s'appeler elle-même.
8. Les variables globales sont utilisées quand de nombreuses fonctions ont besoin d'accéder aux mêmes données. Les variables globales sont très rares en C++. Lorsque vous saurez créer des variables de classe statiques, vous n'avez presque plus jamais besoin de créer des variables globales.
9. La surcharge de fonction est la possibilité d'écrire plusieurs fonctions portant le même nom, mais avec un nombre ou des types de paramètres différents.

Exercices

1. `unsigned long int Perimetre(unsigned short int, unsigned short int);`
2. Voici une réponse possible :

```
unsigned long int Perimetre
     (unsigned short int longueur, unsigned short int largeur)
{
return (2*longueur) + (2*largeur);
}
```

3. La fonction tente de renvoyer une valeur, bien qu'elle soit déclarée comme renvoyant `void` et elle n'y parvient donc pas.
4. Cette fonction serait correcte sans le point-virgule à la fin de la ligne d'en-tête de la définition de fonction.
5. Voici une réponse possible :

```
short int Division(unsigned short int val1,
                   unsigned short int val2)
```

```
{
   if(val2 == 0)
       return -1;
   else
       return val1 / val2;
}
```

6. Voici une solution possible :

```
 1:   #include <iostream>
 2:   using namespace std;
 3:
 4:   short int Division(
 5:      unsigned short int val1,
 6:      unsigned short int val2)
 7:
 8:   int main()
 9:   {
10:     unsigned short int une, deux;
11:     short int reponse;
12:     cout << "Entrez deux nombres. \nPremier : ";
13:     cin >> une;
14:     cout << "Second : ";
15:     cin >> deux;
16:     reponse = Division(une, deux);
17:     if (reponse > -1)
18:        cout << "Reponse : " << reponse;
19:     else
20:        cout << "Erreur, division par zéro impossible !";
21:     return 0;
22:   }
23:
24:  short int Division(unsigned short int val1,
24a:                    unsigned short int val2)
25:  {
26:      if (val2 == 0)
27:          return -1;
28:      else
29:          return val1 / val2;
30:  }
```

7. Voici une réponse possible :

```
1:   #include <iostream>
2:   using namespace std;
3:   typedef unsigned short USHORT;
4:   typedef unsigned long ULONG;
5:
```

```
 6:   ULONG LirePuissance(USHORT n, USHORT puissance);
 7:
 8:   int main()
 9:   {
10:     USHORT nombre, puissance;
11:     ULONG reponse;
12:     cout << "Entrez un nombre: ";
13:     cin >> nombre;
14:     cout << "À quelle puissance ? ";
15:     cin >> puissance;
16:     reponse = LirePuissance(nombre, puissance);
17:     cout << nombre << " à la puissance " << puissance
18:                    << " est égal à "<< reponse << endl;
19:  return 0;
20:  }
21:
22:  ULONG LirePuissance(USHORT n, USHORT puissance)
23:  {
24:     if(puissance == 1)
25:        return n;
26:     else
27:        return(n * LirePuissance(n, puissance-1));
28:  }
```

Chapitre 6

Testez vos connaissances

1. L'opérateur point (`.`) permet d'accéder aux membres d'une classe ou d'une structure.
2. Ce sont les définitions de variables qui réservent la mémoire. Les définitions de classes ne réservent pas de mémoire.
3. La déclaration d'une classe constitue son interface. Elle indique aux clients de cette classe comment interagir avec elle. L'implémentation de la classe est l'ensemble des fonctions membres ; elle est placée en général dans un fichier `.cpp`.
4. Les données membres publiques peuvent être lues par les clients de la classe. On ne peut accéder aux données membres privées que par l'intermédiaire des fonctions membres de la classe.
5. Oui. Les fonctions membres peuvent être privées, même si ce n'est pas indiqué dans ce chapitre. Seules les autres fonctions membres de la classe pourront utiliser la fonction privée.

6. Oui, mais il est préférable de les déclarer privées et de fournir des méthodes publiques d'accès pour ces données.
7. Oui, chaque objet d'une classe possède ses propres données membres.
8. À la différence des définitions de fonctions, les déclarations ont un point-virgule après l'accolade fermante.
9. Voici à quoi ressemble l'en-tête d'une fonction `Miaou()` de la classe `Chat`, qui ne reçoit aucun paramètre et renvoie `void` :

```
void Chat::Miaou()
```

10. Le constructeur est appelé pour initialiser une classe. Cette fonction spéciale porte le même nom que la classe.

Exercices

1. Voici une réponse possible :

```
class Personnel
{
   int Age;
   int Anciennete;
   int Salaire;
};
```

2. Voici une réponse possible. Vous remarquerez que les méthodes d'accès `Get…` ont aussi été rendues constantes, car elles ne changeront rien dans la classe.

```
// Personnel.hpp
class Personnel
{
public:
   int GetAge() const;
   void SetAge(int age);
   int GetAnciennete() const;
   void SetAnciennete(int annees);
   int GetSalaire() const;
   void SetSalaire(int salaire);

private:
   int Age;
   int Anciennete;
   int Salaire;
};
```

3. Voici une réponse possible :

```
 1:   // Personnel.cpp
 2:   #include <iostream>
 3:   #include "Personnel.hpp"
 4:
 5:   int  Personnel::GetAge() const
 6:   {
 7:      return Age;
 8:   }
 9:  void Personnel::SetAge(int age)
10:  {
11:      Age = age;
12:  }
13:  int  Personnel::GetAnciennete() const
14:  {
15:      return Anciennete;
16:  }
17:  void Personnel::SetAnciennete(int annees)
18:  {
19:      Anciennete = annees;
20:  }
21:  int  Personnel::GetSalaire()const
22:  {
23:      return Salaire;
24:  }
25:  void Personnel::SetSalaire(int salaire)
26:  {
27:      Salaire = salaire;
28:  }
29:
30:  int main()
31:  {
32:     using namespace std;
33:
34:     Personnel John;
35:     Personnel Sally;
36:
37:     John.SetAge(30);
38:     John.SetAnciennete(5);
39:     John.SetSalaire(150000);
40:
41:     Sally.SetAge(32);
42:     Sally.SetAnciennete(8);
43:     Sally.SetSalaire(140000);
44:
45:     cout << "Chez Sexiste SA, John et Sally ont
```

```
46:     cout << "le même travail.\n\n";
47:
48:     cout << "John a "<< John.GetAge() << " ans."; << endl;
49:     cout << "Son ancienneté est de " ;
50:     cout << John.GetAnciennete() << " ans." << endl;
51:     cout << " John gagne " << John.GetSalaire();
52:     cout << "€ par an.\n\n";
53:
54:     cout << "Sally, elle, a " << Sally.GetAge();
55:     cout << " ans et son ancienneté est de ";
56:     cout << Sally.GetAnciennete();
57:     cout << " ans. Sally gagne " << Sally.GetSalaire();
58:     cout << " € par an. Sexiste SA porte bien son nom !";
59:  }
```

4. Voici une réponse possible :

```
float Personnel:LireMilliersArrondis()const
{
   return Salaire / 1000;
}
```

5. Voici une réponse possible :

```
class Personnel
{
public:

   Personnel(int age, int annees, int salaire);
   int GetAge() const;
   void SetAge(int age);
   int GetAnciennete() const;
   void SetAnciennete(int annees);
   int GetSalaire() const;
   void SetSalaire(int salaire);


private:
   int Age;
   int Anciennete;
   int Salaire;
};
```

6. Les déclarations de classes doivent se terminer par un point-virgule.

7. La méthode d'accès `GetAge()` est privée. N'oubliez pas que tous les membres d'une classe sont privés, sauf si vous leur attribuez le mot-clé `public`.
8. La variable `saStation` étant privée, vous ne pouvez pas y accéder directement.

   Vous ne pouvez pas appeler `SetStation()` sur la classe, seulement sur des objets.

   Vous ne pouvez pas initialiser `monAutreTele` parce qu'il n'y a pas de constructeur correspondant.

Chapitre 7

Testez vos connaissances

1. Il suffit de séparer les initialisations par des virgules :

```
for (x=0, y=10; x<100; x++, y++)
```

2. Il faut éviter d'utiliser `goto` parce que cette instruction provoque un saut dans n'importe quelle direction et vers une ligne de code quelconque. Le code source devient difficile à comprendre et à mettre à jour.
3. Oui. Si la condition est fausse après l'initialisation, le corps de la boucle `for` ne sera jamais exécuté. Voici un exemple :

```
for (int x=100; x<100; x++)
```

4. La variable `x` est hors de portée, elle n'a donc pas de valeur valide.
5. Oui. Toute boucle peut être imbriquée dans une autre.
6. Voici deux exemples, l'un avec une boucle `for`, l'autre avec une boucle `while` :

```
for(;;)
{
   // cette boucle for ne se termine jamais !
}
while(true)
{
   // cette boucle while ne se termine jamais !
}
```

7. Votre programme semble être "bloqué" car il boucle sans fin, ce qui vous oblige à redémarrer l'ordinateur ou à utiliser des fonctions avancées de votre système d'exploitation pour mettre fin à la tâche.

Exercices

1. Voici une réponse possible :

```
for (int i=0; i<10; i++)
{
   for (int j=0; j<10; j++)
      cout << "0";
   cout << endl;
}
```

2. Voici une réponse possible :

```
for (int x=100; x<=200; x+=2)
```

3. Voici une réponse possible :

```
int x=100;
while (x<200)
   x+=2;
```

4. Voici une réponse possible :

```
int x=100;
do
{
   x+=2;
} while (x<=200);
```

5. Le compteur n'est jamais incrémenté, la boucle `while` ne se terminera jamais.

6. Il y a un point-virgule après la boucle, ce qui correspond à une boucle vide. Le compteur affichera sa valeur après la fin de la boucle `for`.

7. `compteur` est initialisé à 100, mais le test contrôle s'il est inférieur à 10. Le résultat sera donc toujours faux et le corps de la boucle ne s'exécutera jamais. Si vous transformez la première ligne en `int compteur=5;` la boucle s'exécutera tant qu'elle n'a pas atteint la plus petite valeur possible de `int`. Comme `int` est une valeur signée par défaut, ce n'est probablement pas non plus ce qui était prévu.

8. L'instruction `case 0` a certainement besoin d'une instruction `break`. Sinon, elle devrait inclure un commentaire pour être certain que c'est ce qui était prévu.

Chapitre 8

Testez vos connaissances

1. L'opérateur *adresse de* (`&`) permet de déterminer l'adresse d'une variable.
2. C'est l'opérateur d'indirection (`*`) qui permet d'accéder à la valeur stockée à l'adresse contenue dans un pointeur.
3. Un pointeur est une variable qui contient l'adresse d'une autre variable.
4. L'adresse stockée dans le pointeur est l'adresse d'une autre variable. L'opérateur d'indirection (`*`) renvoie la valeur stockée à cette adresse qui est elle-même stockée dans le pointeur.
5. L'opérateur d'indirection renvoie la valeur stockée à l'adresse contenue dans un pointeur. L'opérateur *adresse de* (`&`) renvoie l'adresse mémoire de la variable.
6. `const int * ptrUn` déclare que `ptrUn` est un pointeur vers une constante entière. L'entier lui-même ne peut être modifié avec ce pointeur.

   `int * const ptrDeux` déclare que `ptrDeux` est un pointeur constant vers un entier. Une fois initialisé, il ne pourra pas être réaffecté.

Exercices

1. Effet de ces déclarations :
   a. `int * pUn;` : déclaration d'un pointeur sur un entier.
   b. `int vDeux;` : déclaration d'une variable entière.
   c. `int * pTrois = &vDeux;` : déclaration d'un pointeur sur un entier et initialisation de celui-ci avec l'adresse d'une autre variable, `vDeux`.
2. `unsigned short *pAge=&votreAge;`
3. `*pAge=50;`
4. Voici une réponse possible :

```
1:   #include <iostream>
2:
3:   int main()
4:   {
5:      int unEntier;
6:      int *pEntier = &unEntier;
7:      *pEntier = 5;
8:
9:      std::cout << "Un Entier : "
```

```
10:               << *pEntier << std::endl;
11:
12:     return 0;
13:  }
```

5. `pInt` aurait dû être initialisé. Le plus important est que, n'étant pas initialisé, il pointe sur un emplacement quelconque en mémoire. Si vous stockez un littéral (`9`) à cet emplacement, vous risquez de produire un sérieux bogue.
6. Le programmeur a certainement voulu attribuer `9` à la valeur pointée par `pVar`, ce qui serait une affectation à `Variable`. Malheureusement, `9` a été attribué à `pVar` car l'opérateur d'adressage indirect (`*`) a été oublié. Cette opération risque de provoquer une erreur désastreuse si `pVar` est utilisée pour affecter une valeur, car elle pointe vers ce qui se trouve à l'adresse `9` et non à celle de `Variable`.

Chapitre 9

Testez vos connaissances

1. Une référence est un alias, et un pointeur est une variable qui contient une adresse. Les références ne peuvent être ni nulles (`NULL`), ni affectées.
2. Lorsque l'information sur laquelle on pointe doit être réaffectée, ou si le pointeur peut être nul.
3. Un pointeur nul (`0`).
4. C'est un raccourci pour dire "une référence à un objet constant".
5. Passer *par* référence signifie ne pas faire de copie locale. Cette opération peut être réalisée en passant une référence ou un pointeur.
6. Les trois propositions sont correctes, mais il convient de choisir un style et de le conserver.

Exercices

1. Voici une réponse possible :

```
1:   // Exercice 9.1
2:   #include <iostream>
3:
4:   int main()
5:   {
6:   int varUne = 1;   // initialise varUne à 1
7:   int& rVar = varUne;
```

```
 8:  int* pVar = &varUne;
 9:  rVar = 5;        // initialise varUne à 5
10:  *pVar = 7;       // initialise varUne à 7
11:
12:  // Les trois à venir afficheront :
13:  std::cout << "variable :  " << varUne << std::endl;
14:  std::cout << "reference : " << rVar   << std::endl;
15:  std::cout << "pointeur :   " << *pVar  << std::endl;
16:
17:  return 0;
18:  }
```

2. Voici une réponse possible :

```
1:   int main()
2:   {
3:      int varUne;
4:      const int * const pVar = &varUne;
5:      varUne = 6;
6:      *pVar = 7;
7:      int varDeux;
8:      pVar = &varDeux;
9:   return 0;
10:  }
```

3. Vous ne pouvez pas affecter une valeur à un objet constant et vous ne pouvez pas réaffecter un pointeur constant. Cela signifie que les lignes 6 et 8 posent problème.
4. Voici une réponse possible. Sachez que ce programme est dangereux à cause du pointeur perdu :

```
1:   int main()
2:   {
3:      int * pVar;
4:      *pVar = 9;
5:      return 0;
6:   }
```

5. Voici une réponse possible :

```
1:   int main()
2:   {
3:      int varUne;
4:      int * pVar = &varUne;
5:      *pVar = 9;
6:      return 0;
7:   }
```

6. Voici une réponse possible. Faites attention aux fuites mémoires dans vos programmes :

```
1:   #include <iostream>
2:   int FoncUne();
3:   int main()
4:   {
5:      int localVar = FoncUne();
6:      std::cout << "La valeur de localVar est : " << localVar;
7:      return 0;
8:   }
9:
10:  int FoncUne()
11:  {
12:     int * pVar = new int (5);
13:     return *pVar;
14:  }
```

7. Voici une réponse possible :

```
1:   #include <iostream>
2:   void FoncUne();
3:   int main()
4:   {
5:      FoncUne();
6:      return 0;
7:   }
8:
9:   void FoncUne()
10:  {
11:     int * pVar = new int (5);
12:     cout << "La valeur de *pVar est : " << *pVar;
13:  }
```

8. `CreerChat` renvoie une référence à l'objet `CHAT` créé sur le tas. Puisqu'il n'existe aucun moyen de libérer cet espace mémoire, vous avez produit une fuite mémoire.

9. Voici une réponse possible :

```
1:   #include <iostream>
2:   using namespace std;
3:   class CHAT
4:   {
5:      public :
6:         CHAT(int age) { sonAge = age; }
7:         ~CHAT(){}
8:         int GetAge() const { return sonAge;}
9:      private :
10:        int sonAge;
11:  };
```

```
12:
13:  CHAT * CreerChat(int age);
14:  int main()
15:  {
16:     int age = 7;
17:     CHAT * Caroline = CreerChat(age);
18:     cout << "Caroline a " << Caroline->GetAge() << " ans";
19:     delete Caroline;
20:     return 0;
21:  }
22:
23:  CHAT * CreerChat(int age)
24:  {
25:     return new CHAT(age);
26:  }
```

Chapitre 10

Testez vos connaissances

1. Les fonctions membres surchargées sont des fonctions d'une classe portant le même nom, mais qui diffèrent par le nombre et/ou le type de leurs paramètres.
2. À la différence d'une déclaration, une définition réserve de la mémoire. Les déclarations sont généralement des définitions, à l'exception des déclarations de classe, des prototypes de fonction et des instructions `typedef`.
3. Quand la copie temporaire d'un objet est créée. Cela se produit à chaque fois qu'un objet est passé par valeur.
4. Le destructeur est appelé à chaque fois qu'un objet est détruit, soit parce qu'il devient hors de portée, soit parce que vous avez supprimé son pointeur avec `delete`.
5. L'opérateur d'affectation agit sur un objet existant, alors que le constructeur de copie le duplique.
6. Caché dans chaque fonction membre, le pointeur `this` pointe sur l'objet lui-même.
7. L'opérateur préfixe n'a pas de paramètre. L'opérateur suffixe contient un seul paramètre `int`, qui signale simplement au compilateur qu'il s'agit d'un préfixe.
8. Non. Vous ne pouvez surcharger aucun opérateur pour des types intégrés (ou prédéfinis).
9. On peut le faire, mais c'est fortement déconseillé. La façon dont vous surchargez les opérateurs doit être facilement compréhensible pour ceux qui liront votre code.
10. Aucune. Comme les constructeurs et les destructeurs, ils ne renvoient aucune valeur.

Exercices

1. Voici une réponse possible :

```
class Cercle
{
public:
   Cercle();
   ~Cercle();
   void SetRayon(int);
   int GetRayon();
private:
   int sonRayon;
};
```

2. Voici une implémentation possible du constructeur par défaut :

```
Cercle::Cercle():
   sonRayon(5)
{}
```

3. Voici une réponse possible :

```
Cercle::Cercle(int rayon):
  sonRayon(rayon)
{}
```

4. Voici la création possible des deux opérateurs :

```
const Cercle& Cercle::operator++()
{
   ++(sonRayon);
   return *this;
}

// Opérateur ++(int) suffixe
// extrait puis incrémente
const Cercle Cercle::operator++(int)
{
// déclare le cercle local et l'initialise avec la valeur de *this
Cercle temp(*this);
++(sonRayon);
return temp;
}
```

5. Voici une réponse possible :

```
class Cercle
{
public:
   Cercle();
   Cercle(int);
   ~Cercle();
   void SetRayon(int);
   int GetRayon();
   const Cercle& operator++();
   const Cercle operator++(int);
private:
   int *sonRayon;
};

Cercle::Cercle()
{
   sonRayon = new int(5);
}

Cercle::Cercle(int rayon)
{
   sonRayon = new int(rayon);
}

const Cercle& Cercle::operator++()
{
   ++(*sonRayon);
   return *this;
}
//Opérateur ++(int) suffixe
//extrait puis incrémente
const Cercle Cercle::operator++(int)
{
//déclare le cercle local et l'initialise avec la valeur de *this
Cercle temp(*this);
++(*sonRayon);
return temp;
}
```

6. Voici un constructeur de copie possible :

```
Cercle::Cercle(const Cercle & rhs)
{
   int val = rhs.GetRayon();
   sonRayon = new int(val);
}
```

7. Voici une réponse possible :

```
Cercle& Cercle::operator=(const Cercle & rhs)
{
   if (this == &rhs)
      return *this;
   delete sonRayon;
   sonRayon = new int;
   *sonRayon = rhs.GetRayon();
   return *this;
}
```

8. Voici une réponse possible :

```
 1:  #include <iostream>
 2:  using namespace std;
 3:
 4:  class Cercle
 5:  {
 6:  public:
 7:     //constructeurs
 8:     Cercle();
 9:     Cercle(int);
10:     Cercle(const Cercle &);
11:     ~Cercle() {}
12:
13:  // Méthodes d'accès
14:     void GetRayon(int);
15:     int SetRayon()const;
16:
17:  // opérateurs
18:     const Cercle& operator++();
19:     const Cercle operator++(int);
20:     Cercle& operator=(const Cercle &);
21:
22:  private:
23:     int *sonRayon;
24:  };
25:
26:  Cercle::Cercle()
27:  {sonRayon = new int(5);}
28:
29:  Cercle::Cercle(int rayon)
30:  {sonRayon = new int(rayon);}
31:
32:
33:  Cercle::Cercle(const Cercle & rhs)
```

```
34:  {
35:     int val = rhs.GetRayon();
36:     sonRayon = new int(val);
37:  }
38:
39:  Cercle& Cercle::operator=(const Cercle & rhs)
40:  {
41:     if (this == &rhs)
42:        return *this;
43:     *sonRayon = rhs.GetRayon();
44:     return *this;
45:  }
46:
47:  const Cercle& Cercle::operator++()
48:  {
49:     ++(*sonRayon);
50:     return *this;
51:  }
52:
53:  // Opérateur ++(int) suffixe
54:  // extrait puis incrémente
55:  const Cercle Cercle::operator++(int)
56:  {
57:  // déclare le cercle local et l'initialise avec la valeur de *this
58:  Cercle temp(*this);
59:  ++(*sonRayon);
60:  return temp;
61:  }
62:  int Cercle::LireRayon() const
63:  {
64:     return *sonRayon;
65:  }
66:  int main()
67:  {
68:     Cercle CercleUn, CercleDeux(9);
69:     CercleUn++;
70:     ++CercleDeux;
71:     cout << "CercleUn : " <<CercleUn.GetRayon() << endl;
72:     cout << "CercleDeux : " <<CercleDeux.GetRayon() << endl;
73:     CercleUn = CercleDeux;
74:     cout << "CercleUn : " <<CercleUn.GetRayon() << endl;
75:     cout << "CercleDeux : " <<CercleDeux.GetRayon() << endl;
76:  return 0;
77:  }
```

9. Vous devez contrôler si `rhs` est égal à `this`.

10. Cet `operator+` modifie la valeur d'un des opérandes plutôt que de créer un nouvel objet `EntierCourt` avec la somme. Voici le code corrigé :

```
EntierCourt EntierCourt::operator+ (const EntierCourt& rhs)
{
   return EntierCourt(saVal + rhs.GetVal());
}
```

Chapitre 11

Testez vos connaissances

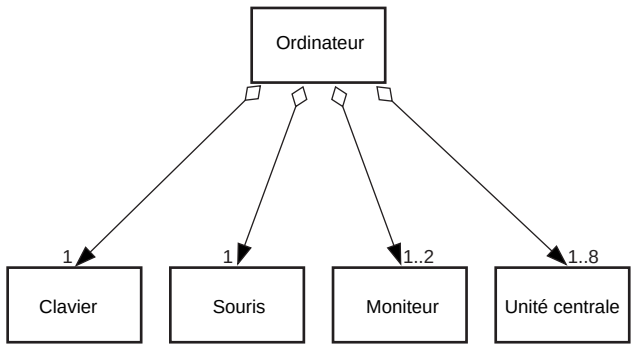
1. La programmation procédurale met l'accent sur les fonctions qui sont séparées des données. La programmation orientée objet associe les données et les fonctions, les intègre dans les objets et s'appuie sur les interactions entre ces objets.
2. Les phases d'analyse et de conception orientées objet sont la conceptualisation (simple phrase décrivant l'objectif du projet), l'analyse (processus de compréhension des besoins), et la conception (processus de création du modèle de vos classes), permettant de générer le code. Elles sont suivies de l'implémentation, des tests et du déploiement.
3. L'encapsulation consiste à réunir en une classe toutes les données et les fonctionnalités d'une entité.
4. Un domaine est un secteur d'activité pour lequel vous créez un produit.
5. Un acteur est une personne ou un système externe au système développé ; il interagit avec le système que vous développez.
6. Un cas d'utilisation est une description de l'utilisation du logiciel. C'est la description d'une interaction entre un acteur et le système lui-même.
7. A est vrai, B ne l'est pas.

Exercices

1. Le diagramme suivant propose une réponse possible :
2. Les voitures, les motos, les camions, les vélos, les piétons et les véhicules prioritaires empruntent tous le carrefour. Des feux de circulation sont présents avec un signal lumineux réservé aux piétons.

   La surface de la route devrait être incluse dans cette simulation, car son état peut influencer le trafic, mais nous laisserons cette question de côté pour cette première conception.

Figure C.01



L'objet le plus naturel est le carrefour lui-même. Il pourrait mettre à jour une liste des voitures qui attendent pour passer dans chaque direction et une liste des piétons qui attendent pour traverser sur le passage réservé. Il aura besoin de méthodes pour sélectionner les véhicules ou les piétons autorisés à traverser.

Il n'y a qu'un carrefour, vous devez donc chercher comment vous assurer de l'instanciation d'un seul objet (pensez aux méthodes statiques et à l'accès protégé).

Les piétons et les voitures sont les clients de l'intersection. Ils ont plusieurs caractéristiques communes : ils peuvent apparaître à tout moment, leur nombre est aléatoire, et ils attendent le signal pour passer (bien que ce soit à des endroits différents). Vous pouvez donc envisager une classe de base commune pour les piétons et les voitures.

Les classes pourraient donc être les suivantes :

```
class Entite;   // un client de l'intersection

// la racine de toutes les voitures, camions, cycles
// et véhicules prioritaires.
class Vehicule: Entite ...;

// la racine de tous les piétons
class Pieton : Entite...;

class Voiture: public Vehicule...;
class Camion: public Vehicule...;
class Moto: public Vehicule...;
class Velo: public Vehicule...;
class Vehicule_Prioritaire : public Vehicule...;

// contient les listes de voitures et de piétons
// qui attendent pour passer
class Intersection;
```

Chapitre 12

Testez vos connaissances

1. Une v-table, ou table de fonction virtuelle, est un moyen couramment utilisé par les compilateurs pour gérer les fonctions virtuelles en C++. Cette table contient la liste des adresses de toutes les fonctions virtuelles et, selon le type de l'objet pointé au moment de l'exécution, elle appelle la fonction appropriée.
2. Tout destructeur de classe peut être déclaré virtuel. Lorsque le pointeur est supprimé, le type l'objet au moment de l'exécution est évalué et le destructeur dérivé adéquat est appelé.
3. Il s'agissait d'une question piége : il n'y a pas de constructeurs virtuels.
4. Vous pouvez le faire, en créant dans votre classe une méthode virtuelle qui, elle-même, appelle le constructeur de copie.
5. `Base::NomFonction();`
6. `NomFonction();`
7. Oui, la virtualité est héritée et il est *impossible* de la contourner.
8. Les membres `protected` sont accessibles aux fonctions membres des objets dérivés.

Exercices

1. `virtual void UneFonction(int);`
2. Puisque vous présentez une déclaration de `Carre`, vous n'avez pas à vous occuper de `Forme` puisqu'elle est automatiquement incluse dans `Rectangle`.

```
class Carre : public Rectangle
{};
```

3. Comme pour l'Exercice 2, ne vous occupez pas de `Forme` :

```
Carre::Carre(int longueur):
   Rectangle(longueur, longueur){}
```

4. Voici une réponse possible :

```
class Carre
{
  public:
```

```
//...
virtual Carre * clone() const { return new Carre(*this); }
//...
};
```

5. Ce code ne contient peut-être pas d'erreur. `Fonction` attend un objet `Forme`. Vous lui avez passé un `Rectangle` "réduit" en `Forme`. Tant que vous n'avez pas besoin des parties de `Rectangle`, tout ira bien. Sinon, vous devrez modifier `Fonction` pour qu'elle prenne en paramètre un pointeur ou une référence sur un objet `Forme`.
6. Vous ne pouvez pas déclarer virtuel un constructeur de copie.

Chapitre 13

Testez vos connaissances

1. `Tableau[0]` et `Tableau[24]`.
2. Il faut ajouter un niveau d'indice pour chaque dimension. `Tableau[2] [3] [2]`, par exemple, est un tableau à trois dimensions. La première dimension contient deux éléments, la deuxième trois et la troisième deux.
3. Voici l'initialisation du tableau précédent :

```
Tableau[2][3][2]= { { {1, 2}, {3, 4}, {5, 6} },
                    { {7, 8}, {9, 10}, {11, 12} } };
```

4. 10 * 5 * 20 = 1 000.
5. Les tableaux et les listes chaînées sont des conteneurs d'informations. Les listes chaînées peuvent cependant être liées ensemble en fonction des besoins.
6. Cette chaîne contient 16 caractères : le dernier que vous voyez est le quinzième et le caractère nul termine la chaîne.
7. Le caractère nul.

Exercices

1. Voici une solution possible. Votre tableau pourrait avoir un autre nom, mais il doit être suivi de `[3][3]` pour contenir un échiquier de 3 sur 3 :

```
int Echiquier[3][3];
```

2. `int Echiquier[3] [3] = { {0, 0, 0}, {0, 0, 0}, {0, 0, 0} };`

3. Voici une solution possible. Elle utilise les fonctions `strcpy()` et `strlen()` :

```
#include <iostream>
#include <string.h>
using namespace std;

int main()
{
   char prenom[] = "Alfred";
   char initiale[] = "E";
   char nom[] = "Numan";
   char complet[80];
   int  offset = 0;

   strcpy(complet, prenom);
   offset = strlen(prenom);
   strcpy(complet + offset, " ");
   offset += 1;
   strcpy(complet + offset, initiale);
   offset += strlen(initiale);
   strcpy(complet + offset, ". ");
   offset += 2;
   strcpy(complet + offset, nom);

   cout << prenom << "-" << initiale << "-"
        << nom << endl;
   cout << "Nom complet : " << complet << endl;

   return 0;
}
```

4. La taille du tableau est de cinq éléments par quatre éléments, mais le code en initialise 4 * 5.
5. Vous vouliez écrire `i < 5`, mais vous avez écrit `i <= 5`. Ce code fonctionnera pour les valeurs `i == 5` et `j == 4`, or l'élément `Tableau[5][4]` n'existe pas.

Chapitre 14

Testez vos connaissances

1. Un transtypage descendant déclare qu'un pointeur d'une classe de base doit être considéré comme le pointeur d'une classe dérivée.
2. Cela fait référence au déplacement des fonctionnalités partagées dans une classe de base commune, plus haut dans la hiérarchie. Si plusieurs classes partagent une

méthode, il est souhaitable de trouver une classe de base commune dans laquelle placer cette méthode.

3. Si ni l'une ni l'autre des classes n'hérite avec le mot-clé `virtual`, deux `Formes` sont créés, une pour `Rectangle` et une autre pour `Forme`. Si le mot-clé `virtual` est utilisé pour chaque classe, il n'y aura qu'une seule `Forme` créée.
4. `Cheval` et `Oiseau` initialisent leur classe de base, `Animal`, dans leur constructeur. `Pegase` le fait également ; lorsqu'un `Pegase` est créé, les initialisations d'`Animal` pour un `Cheval` et un `Oiseau` sont ignorées.
5. Voici une réponse possible :

```
class Vehicule
{
   virtual void Deplacer() = 0;
}
```

6. Aucune ne doit être redéfinie, sauf si vous voulez une classe qui ne soit pas abstraite. Dans ce cas, les trois devront être redéfinies.

Exercices

1. `class Jet : public Fusee, public Avion`
2. `class Boeing : public Jet`
3. Voici une réponse possible :

```
class Vehicule
{
   virtual void Deplacer() = 0;
   virtual void Transporter() = 0;
};
class Auto : public Vehicule
{
   virtual void Deplacer();
   virtual void Transporter();
};
class Bus : public Vehicule
{
   virtual void Deplacer();
   virtual void Transporter();
};
```

4. Voici une réponse possible :

```
class Vehicule
```

```
{
   virtual void Deplacer() = 0;
   virtual void Transporter() = 0;
};
class Auto : public Vehicule
{
   virtual void Deplacer();
};
class Bus : public Vehicule
{
   virtual void Deplacer();
   virtual void Transporter();
};
class Formule1 : public Auto
{
   virtual void Transporter();
};
class Coupe : public Auto
{
   virtual void Transporter();
};
```

Chapitre 15

Testez vos connaissances

1. Oui, ce sont des variables membres et leur accès peut être contrôlé comme pour toute autre variable. Si elles sont privées, on ne peut y accéder qu'avec des fonctions membres ou, plus couramment, avec des fonctions membres statiques.
2. `static int saStatique;`
3. `static int Fonction();`
4. `long (* fonction) (int);`
5. `long ( Auto::*fonction) (int);`
6. `long ( Auto::*fonction) (int) leTableau [10];`

Exercices

1. Voici une réponse possible :

```
0:   // Ex1501.cpp
1:   class maClasse
2:   {
```

```
 3:  public:
 4:     maClasse();
 5:     ~maClasse();
 6:  private:
 7:     int sonMembre;
 8:     static int saStatique;
 9:  };
10:
11:  maClasse::maClasse():
12:     sonMembre(1)
13:  {
14:     saStatique++;
15:  }
16:
17:  maClasse::~maClasse()
18:  {
19:     saStatique--;
20:  }
21:
22:  int maClasse::saStatique = 0;
23:
24:  int main()
25:  {
26:     // faire quelque chose
27:     return 0;
28:  }
```

2. Voici une réponse possible :

```
0:   // Ex1502.cpp
1:   #include <iostream>
2:   using namespace std;
3:   class maClasse
4:   {
5:   public:
6:      maClasse();
7:      ~maClasse();
8:      void AfficherMembre();
9:      void AfficherStatique();
10:  private:
11:     int sonMembre;
12:     static int saStatique;
13:  };
14:
15:  maClasse::maClasse():
16:     sonMembre(1)
17:  {
```

```
18:     saStatique++;
19:  }
20:
21:  maClasse::~maClasse()
22:  {
23:     saStatique--;
24:     cout << "Dans le destructeur. saStatique : "
24a:         << saStatique << endl;
25:  }
26:
27:  void maClasse::AfficherMembre()
28:  {
29:     cout << "sonMembre : " << sonMembre << endl;
30:  }
31:
32:  void maClasse::AfficherStatique()
33:  {
34:     cout << "saStatique : " << saStatique << endl;
35:  }
36:  int maClasse::saStatique = 0;
37:
38:  int main()
39:  {
40:     maClasse obj1;
41:     obj1.AfficherMembre();
42:     obj1.AfficherStatique();
43:
44:     maClasse obj2;
45:     obj2.AfficherMembre();
46:     obj2.AfficherStatique();
47:
48:     maClasse obj3;
49:     obj3.AfficherMembre();
50:     obj3.AfficherStatique();
51:     return 0;
52:  }
```

3. Voici une solution possible :

```
0:   // Ex1503.cpp
1:   #include <iostream>
2:   using namespace std;
3:   class maClasse
4:   {
5:   public:
6:      maClasse();
7:      ~maClasse();
```

```
 8:     void AfficherMembre();
 9:     static int GetStatique();
10:  private:
11:     int sonMembre;
12:     static int saStatique;
13:  };
14:
15:  maClasse::maClasse():
16:     sonMembre(1)
17:  {
18:     saStatique++;
19:  }
20:
21:  maClasse::~maClasse()
22:  {
23:     saStatique--;
24:     cout << "Dans le destructeur. saStatique : "
24a:         << saStatique << endl;
25:  }
26:
27:  void maClasse::AfficherMembre()
28:  {
29:     cout << "sonMembre : " << sonMembre << endl;
30:  }
31:
32:  int maClasse::saStatique = 0;
33:
34:  int maClasse::GetStatique()
35:  {
36:     return saStatique;
37:  }
38:
39:  int main()
40:  {
41:     maClasse obj1;
42:     obj1.AfficherMembre();
43:     cout << "Statique : " << maClasse::GetStatique() << endl;
44:
45:     maClasse obj2;
46:     obj2.AfficherMembre();
47:     cout << "Statique : " << maClasse::GetStatique() << endl;
48:
49:     maClasse obj3;
50:     obj3.AfficherMembre();
51:     cout << "Statique : " << maClasse::GetStatique() << endl;
52:     return 0;
53 :  }
```

4. Voici une réponse possible :

```
 0:  // Ex1504.cpp
 1:  #include <iostream>
 2:  using namespace std;
 3:  class maClasse
 4:  {
 5:  public:
 6:     maClasse();
 7:     ~maClasse();
 8:     void AfficherMembre();
 9:     static int GetStatique();
10:  private:
11:     int sonMembre;
12:     static int saStatique;
13:  };
14:
15:  maClasse::maClasse():
16:     sonMembre(1)
17:  {
18:     saStatique++;
19:  }
20:
21:  maClasse::~maClasse()
22:  {
23:    saStatique--;
24:     cout << "Dans le destructeur. saStatique : "
24a:         << saStatique << endl;
25:  }
26:
27:  void maClasse::AfficherMembre()
28:  {
29:     cout << "sonMembre : " << sonMembre << endl;
30:  }
31:
32:  int maClasse::saStatique = 0;
33:
34:  int maClasse::GetStatique()
35:  {
36:     return saStatique;
37:  }
38:
39:  int main()
40:  {
41:     void (maClasse::*PFM) ();
42:
43:     PFM = maClasse::AfficherMembre;
```

```
44:
45:     maClasse obj1;
46:     (obj1.*PFM) ();
47:     cout << "Statique : " << maClasse::GetStatique() << endl;
48:
49:     maClasse obj2;
50:     (obj2.*PFM)();
51:     cout << "Statique : " << maClasse::GetStatique() << endl;
52:
53:     maClasse obj3;
54:     (obj3.*PFM)();
55:     cout << "Statique : " << maClasse::GetStatique() << endl;
56:     return 0;
57: }
```

5. Voici une réponse possible :

```
 0: // Ex1505.cpp
 1: #include <iostream>
 2: using namespace std;
 3: class maClasse
 4: {
 5: public:
 6:    maClasse();
 7:    ~maClasse();
 8:    void AfficherMembre();
 9:    void AfficherSecond();
10:    void AfficherTroisieme();
11:    static int GetStatique();
12: private:
13:    int sonMembre;
14:    int sonSecond;
15:    int sonTroisieme;
16:    static int saStatique;
17: };
18:
19: maClasse::maClasse():
20:    sonMembre(1),
21:    sonSecond(2),
22:    sonTroisieme(3)
23: {
24:    saStatique++;
25: }
26:
27: maClasse::~maClasse()
28: {
29:    saStatique--;
```

```
30:     cout << "Dans le destructeur. saStatique : "
30a:         << saStatique << endl;
31:  }
32:
33:  void maClasse::AfficherMembre()
34:  {
35:     cout << "sonMembre : " << sonMembre << endl;
36:  }
37:
38:  void maClasse::AfficherSecond()
39:  {
40:     cout << "sonSecond : " << sonSecond << endl;
41:  }
42:
43:  void maClasse::AfficherTroisieme()
44:  {
45:     cout << "sonTroisieme : " << sonTroisieme << endl;
46:  }
47:  int maClasse::saStatique = 0;
48:
49:  int maClasse::GetStatique()
50:  {
51:     return saStatique;
52:  }
53:
54:  int main()
55:  {
56:     void (maClasse::*PFM) ();
57:
58:     maClasse obj1;
59:     PFM = maClasse::AfficherMembre;
60:     (obj1.*PFM) ();
61:     PFM = maClasse::AfficherSecond;
62:     (obj1.*PFM) ();
63:     PFM = maClasse::AfficherTroisieme;
64:     (obj1.*PFM) ();
65:     cout << "Statique : " << maClasse::GetStatique() << endl;
66:
67:     maClasse obj2;
68:     PFM=maClasse::AfficherMembre;
69:     (obj2.*PFM)();
70:     PFM=maClasse::AfficherSecond;
71:     (obj2.*PFM)();
```

```
72:     PFM=maClasse::AfficherTroisieme;
73:     (obj2.*PFM)();
74:     cout << "Statique : " << maClasse::GetStatique() << endl;
75:
76:     maClasse obj3;
77:     PFM=maClasse::AfficherMembre;
78:     (obj3.*PFM)();
79:     PFM=maClasse::AfficherSecond;
80:     (obj3.*PFM)();
81:     PFM=maClasse::AfficherTroisieme;
82:     (obj3.*PFM)();
83:     cout << "Statique : " << maClasse::GetStatique() << endl;
84:
85:     return 0;
86: }
```

Chapitre 16

Testez vos connaissances

1. Une relation *est-un* est établie avec un héritage multiple.
2. Une relation *a-un* est établie avec l'agrégation (la composition) ; une classe possède un membre qui est un objet d'un autre type.
3. L'agrégation correspond à l'idée d'une classe ayant une donnée membre qui est un objet d'un autre type. La délégation illustre le concept qu'une classe utilise une autre classe pour réaliser une tâche.
4. La délégation illustre l'idée qu'une classe utilise une autre classe pour réaliser une tâche. "Implémenté en terme de" illustre l'idée de l'héritage de l'implémentation d'une autre classe.
5. Une fonction amie est une fonction qui peut accéder aux membres privés et protégés de votre classe.
6. Une classe amie est une classe dont toutes les fonctions membres sont des fonctions amies de votre classe.
7. Non, cette relation n'est pas commutative.
8. Non, cette relation ne se transmet pas par héritage.
9. Non, cette relation n'est pas associative.
10. N'importe où dans la déclaration de la classe.

Exercices

1. Voici une déclaration possible pour `Animal` :

```
class Animal:
{
private:
   String sonNom;
};
```

2. Voici une réponse possible :

```
class Limite : public Tableau
{
// ...
}
```

3. Voici une réponse possible :

```
class Ensemble : private Tableau
{
// ...
}
```

4. Voici un programme possible :

```
 0:       #include <iostream.h>
 1:       #include <string.h>
 2:
 3:       class String
 4:       {
 5:          public:
 6:              // constructeurs
 7:              String();
 8:              String(const char *const);
 9:              String(const String &);
10:             ~String();
11:
12:             // opérateurs surchargés
13:             char & operator[](int indice);
14:             char operator[](int indice) const;
15:             String operator+(const String&);
16:             void operator+=(const String&);
17:             String & operator= (const String &);
18:             friend ostream& operator<<( ostream& leFlux,
19:                                         String& laChaine);
20:             friend istream& operator>>( istream& leFlux,
```

```
21:                                          String& laChaine);
22:            // méthodes d'accès
23:            int GetLongueur()const { return saLongueur; }
24:            const char * GetString() const
24a:                { return saString; }
25:            // static int CpteConstructeur;
26:
27:         private:
28:            String (int);         // constructeur privé
29:            char * saString;
30:            unsigned short saLongueur;
31:
32:      };
33:
34:      ostream& operator<<( ostream& leFlux, String& laChaine)
35:      {
36:          leFlux << laChaine.GetString();
37:          return leFlux;
38:      }
39:
40:      istream& operator>>( istream& leFlux,String& laChaine)
41:      {
42:          leFlux >> laChaine.GetString();
43:          return leFlux;
44:      }
45:
46:      int main()
47:      {
48:         String laChaine("Bonjour.");
49:         cout << laChaine;
50:    return 0;
51: }
```

5. Vous ne pouvez pas placer la déclaration `friend` dans la fonction. Vous devez déclarer la fonction comme une fonction amie dans la classe.

6. Voici le listing corrigé :

```
0:   // Cherchez l'erreur
1:   #include <iostream>
2:   using namespace std;
3:   class Animal;
4:
5:   void SetVal(Animal& , int);
6:
7:   class Animal
8:   {
9:   public:
```

```
10:     friend void SetVal (Animal&, int);
11:     int GetPoids()const { return sonPoids; }
12:     int GetAge() const { return sonAge; }
13:  private
14:     int sonPoids;
15:     int sonAge;
16:  };
17:
18:  void SetVal (Animal& unAnimal, int lePoids)
19:  {
20:     unAnimal.sonPoids = lePoids;
21:  }
22:
23:  int main()
24:  {
25:     Animal medor;
26:     SetVal(medor, 5);
27:     return 0;
28:  }
```

7. La fonction `SetVal(Animal&, int)` a été déclarée comme amie, mais pas la fonction surchargée `SetVal(Animal&, int, int)`.

8. Voici le listing corrigé :

```
 0:   /// Cherchez l'erreur
 1:   #include <iostream>
 2:   using namespace std;
 3:   class Animal;
 4:
 5:   void SetVal(Animal& , int);
 6:   void SetVal(Animal&, int, int); // voici la modification !
 7:
 8:   class Animal
 9:   {
10:   friend void SetVal(Animal& , int);
11:   friend void SetVal(Animal&, int, int);
12:   private
13:      int sonPoids;
14:      int sonAge;
15:   };
16:
17:   void SetVal(Animal& unAnimal, int lePoids)
18:   {
19:     unAnimal.sonPoids = lePoids;
20:   }
21:
22:   void SetVal(Animal& unAnimal, int lePoids, int unAge)
```

```
23:   {
24:      unAnimal.sonPoids = lePoids;
25:      unAnimal.sonAge = unAge;
26:   }
27:
28:   int main()
29:   {
30:      Animal medor;
31:      SetVal(medor, 5);
32:      SetVal(medor,7,9);
33:      return 0;
34:   }
```

Chapitre 17

Testez vos connaissances

1. L'opérateur d'insertion (<<) est un opérateur membre de l'objet `ostream`. Il permet d'écrire vers l'unité de sortie.
2. L'opérateur d'extraction (>>) est un opérateur membre de l'objet `istream`. Il permet d'écrire dans les variables de votre programme.
3. La première forme de `get()` n'a pas de paramètres. Elle renvoie la valeur du caractère trouvé ou `EOF` (fin de fichier) si l'on a atteint la fin de fichier.

   La deuxième forme de `get()` reçoit comme paramètre une référence de caractère. Ce caractère est rempli avec le caractère suivant du flux d'entrée. La valeur renvoyée est un objet `iostream`.

   La troisième forme de `get()` reçoit un tableau, un nombre maximal de caractères à récupérer et un caractère de fin. Cette forme de `get()` remplit le tableau avec un nombre de caractères égal au nombre d'éléments du tableau moins un, sauf si elle rencontre un caractère de fin. Dans ce cas elle écrit immédiatement un caractère `null` en laissant le caractère de fin dans le tampon.
4. `cin.read()` permet de lire des structures de données binaires.

   `getline()` permet de lire à partir du tampon de `istream`.
5. Largeur nécessaire pour afficher le nombre entier.
6. Une référence à un objet `istream`.
7. Nom du fichier à ouvrir.
8. `ios::ate` vous positionne en fin de fichier, mais vous pouvez écrire des données n'importe où dans le fichier.

Exercices

1. Voici une réponse possible :

```
 0:     // Ex1701.cpp
 1:     #include <iostream>
 2:     int main()
 3:     {
 4:        int x;
 5:        std::cout << "Entrez un nombre : ";
 6:        std::cin >> x;
 7:        std::cout << "Vous avez tapé : " << x << std::endl;
 8:        std::cerr << "...destine a cerr !" << std::endl;
 9:        std::clog << "...destine a clog !" << std::endl;
10:     return 0;
11:    }
```

2. Voici une réponse possible :

```
0:   // Ex1702.cpp
1:   #include <iostream>
2:   int main()
3:   {
4:      char nom[80];
5:      std::cout << "Entrez votre nom : ";
6:      std::cin.getline(nom, 80);
7:      std::cout << "Vous avez tapé : " << nom << std::endl;
8:      return 0;
9:   }
```

3. Voici une réponse possible :

```
 0:     // Ex1703.cpp
 1:     #include <iostream>
 2:     using namespace std;
 3:
 4:     int main()
 5:     {
 6:        char ch;
 7:        cout << "Entrez une phrase: ";
 8:        while ( cin.get(ch) )
 9:        {
10:          switch (chaîne)
11:             {
12:               case '!':
13:             cout << '$';
14:               break;
```

```
15:             case '#':
16:               break;
17:              default:
18:                cout << ch;
19:                break;
20:              }
21:        }
22:     return 0;
23:     }
```

4. Voici une réponse possible :

```
 0: //Ex1704.cpp
 1: #include <fstream>
 2: #include <iostream>
 3: using namespace std;
 4:
 5: int main(int argc, char**argv)   // renvoie 1 si erreur
 6: {
 7:   if (argc != 2)
 8:   {
 9:      cout << "Usage: argv[0] <fichier>\n";
10:      return(1);
11:   {
12:
13:   // ouverture du flux d'entrée
14:   ifstream fin (argv[1], ios::binaire);
15:   if (!fin)
16:   {
17:      cout << "Impossible d'ouvrir " << argv[1]
17a:          << "en lecture.\n";
18:      return (1);
19:   }
20
21    char ch;
22:   while (fin.get(ch))
23:      if ((ch > 32 && ch < 127) || ch == '\n' || ch == '\t')
24:         cout << ch;
25:    fin.close();
26 }
```

5. Voici une réponse possible :

```
0:   // Ex1705.cpp
1:   #include <iotream>
2:
3:   int main(int argc, char**argv)  // renvoie 1 si erreur
```

```
4:   {
5:      for (int ctr = argc-1; ctr>0 ; ctr--)
6:           std::cout << argv[ctr] << " ";
7:   }
```

Chapitre 18

Testez vos connaissances

1. `Externe::Interne::MaFonction();`.
2. Lorsque le listing atteindra `ICI`, ce sera la version globale de `X` qui sera utilisée, donc 4.
3. Oui, vous pouvez utiliser des noms définis dans un espace de noms en les qualifiant à l'aide de l'espace de noms.
4. Dans un espace de noms normal, les noms sont utilisables en dehors de l'unité où l'espace de noms est déclaré. Dans un espace de noms anonyme, leur utilisation est limitée à l'unité où l'espace de noms est déclaré.
5. Le mot-clé `using` peut servir pour les directives et les déclarations. Une directive `using` permet d'utiliser tous les noms d'un espace de noms comme s'ils étaient normaux. Une déclaration `using` permet par contre au programme d'utiliser un nom d'un espace de noms sans le qualifier.
6. Les espaces de noms anonymes sont des espaces n'ayant pas de noms. Ils permettent d'envelopper une collection de déclarations pour les protéger de conflits de noms éventuels. Dans ce cas, les noms ne peuvent pas être utilisés en dehors de l'unité dans laquelle est déclaré l'espace de noms.
7. L'espace de noms standard `std` est défini par la bibliothèque C++ standard. Elle comprend les déclarations de tous les noms répertoriés dans la bibliothèque standard.

Exercices

1. Le fichier en-tête standard C++ `iostream` déclare `cout` et `endl` dans l'espace de noms `std`. Ils ne peuvent être utilisés en dehors de `std` que s'ils sont qualifiés à l'aide de cet espace de noms.
2. Vous pouvez ajouter la ligne suivante entre les lignes 0 et 1 :

   ```
   using namespace std;
   ```

   Vous pouvez ajouter les deux lignes suivantes entre les lignes 0 et 1 :

```
using std::cout;
using std::endl;
```

Vous pouvez modifier la ligne 3 comme suit :

```
std::cout << "Bonjour !" << std::endl;
```

3. Voici une réponse possible :

```
Namespace MesElements
{
   class MaClasse
   {
      // élements de MaClasse
   }
}
```

Chapitre 19

Testez vos connaissances

1. Les modèles sont intégrés au langage C++ et le type des données est contrôlé. Les macros sont implémentées par le processeur et n'effectuent aucun contrôle de type.
2. Le paramètre d'un modèle crée une instance du modèle pour chaque type. Si vous créez six instances de modèle, six classes ou fonctions différentes seront créées. Les paramètres d'une fonction modifient le comportement ou les données de la fonction et une seule fonction est créée.
3. Une fonction amie modèle général crée une fonction pour chaque type de la classe paramétrée. La fonction de type spécifique crée une instance de type spécifique pour chaque instance de la classe paramétrée.
4. Oui, en créant une fonction spécialisée pour l'instance considérée. Outre la création de `Array<t>::UneFonction()`, créez également `Array<int>::UneFonction()` pour modifier le comportement des tableaux d'entiers.
5. Une pour chaque type d'instance de la classe.
6. La classe doit définir un constructeur par défaut ainsi qu'un opérateur d'affectation surchargé.
7. STL signifie *Standard Template Library* (bibliothèque de modèles standard). Cette bibliothèque est importante, car elle contient plusieurs classes modèles déjà créées et prêtes à l'emploi. Puisqu'elles font partie du standard C++, n'importe quel compilateur conforme au standard les acceptera.

Exercices

1. Voici une façon d'implémenter ce modèle :

```
 0:   //Exercice 19.1
 1:   template <class type>
 2:   class Liste
 3:   {
 4:
 5:   public:
 6:      Liste():tete(0),fin(0),leNbre(0)  { }
 7:      virtual -Liste();
 8:
 9:      void insere( Type valeur );
10:     void ajoute( Type valeur );
11:     int present( Type valeur ) const;
12:     int vide() const { return tete == 0; }
13:     int nbre() const { return leNbre; }
14:
15:  private:
16:     class ListeCellule
17:     {
18:        public:
19:         ListeCellule(Type val, ListeCellule *cell = 0):
19a:          val(valeur), suivante(cell) {}
20:           Type val;
21:           ListeCellule *suivante;
22:       };
23:
24:       ListeCellule *tete;
25:       ListeCellule *fin;
26:       int leNbre;
27:  };
```

2. Voici une réponse possible :

```
0:   // Exercice 19.2
1:   void List::insere(int valeur)
2:   {
3:      ListeCellule *pt = new ListeCellule ( valeur, tete );
4:
5:   // cette ligne est ajoutée pour gérer fin
6:   if (tete == 0) fin = pt;
7:
8:      tete = pt;
9:      leNbre++;
```

```
10:  }
11:
12:  void List::ajoute(int valeur)
13:  {
14:     ListeCellule *pt = new ListeCellule( valeur );
15:     if ( tete == 0 )
16:          tete = pt;
17:     else
18:          fin -> suivante = pt;
19:
20:     fin = pt;
21:     leNbre++;
22:  }
23:
24:  int List::present( int valeur ) const
25:  {
26:     if (tete == 0 ) return 0;
27:     if (tete->val == valeur || fin->val == valeur )
28:         return 1;
29:
30:     ListeCellule *pt = tete->suivante;
31:     for (; pt != fin; pt = pt ->suivante)
32:        if ( pt->val == valeur )
33:          return 1;
34:
35:     return 0;
36:  }
```

3. Voici une réponse possible :

```
 0:  // Exercice 19.3
 1:  template <class Type>
 2:  Liste<Type>::~Liste()
 3:  {
 4:     ListeCellule *pt = tete;
 5:
 6:     while ( pt )
 7:     {
 8:        ListeCellule *tmp = pt;
 9:        pt = pt->suivante;
10:        delete tmp;
11:     }
12:     tete = fin = 0;
13:  }
14:
15:  template <class Type>
16:  void Liste<Type>::insere (Type valeur)
```

```
17:  {
18:  ListeCellule *pt = new ListeCellule ( valeur, tete );
19:     assert (pt !=0);
20:
21:     // cette ligne est ajoutée pour gérer fin
22:     if (tete == 0) fin = pt;
23:
24:     tete = pt;
25:     leNbre++;
26:  }
27:
28:  template <class Type>
29:  void Liste<Type>::ajoute( Type valeur )
30:  {
31:     ListeCellule *pt = new ListeCellule( valeur );
32:     if (tete == 0)
33:         tete = pt;
34:     else
35:         fin->suivante = pt;
36:
37:     fin = pt;
38:     leNbre++;
39:  }
40:
41:  template <class Type>
42:  int Liste<Type>::present( Type valeur ) const
43:  {
44:     if (tete == 0) return 0;
45:     if (tete->val == valeur || fin->val == valeur)
46:        return 1;
47:
48:     ListeCellule *pt = tete->suivante;
49:     for (; pt != fin; pt = pt ->suivante)
50:         if ( pt -> val == valeur )
51:           return 1;
52:
53:     return 0;
54:  }
```

4. Voici la déclaration des trois objets :

```
Liste<String> ListeChaines;
Liste<Chat> ListeChats;
Liste<int> ListeEntiers;
```

5. `Chat` n'ayant pas défini l'opérateur ==, toutes les opérations qui comparent les valeurs dans la liste de cellules, comme `present()`, vont provoquer des erreurs de compilation.

Pour éviter ce genre de situation, n'hésitez pas à bien commenter avant la définition du modèle, pour indiquer les opérations à définir afin que l'instanciation puisse être compilée.

6. Voici une réponse possible :

```
friend int operator==( const Type& lhs, const Type& rhs );
```

7. Voici une réponse possible :

```
 0:  //Exercice 19.7
 1:  template <class Type>
 2:  int Liste<Type>::operator==( const Type& lhs,
2a:                               const Type& rhs)
 3:  {
 4:    // on commence par comparer les longueurs
 5:    if (lhs.leNbre != rhs.leNbre)
 6:       return 0;  // longueurs différentes
 7:
 8:    ListeCellule *lh = lhs.tete;
 9:    ListeCellule *rh = rhs.tete;
10:
11:    for(; lh != 0; lh = lh.suivante, rh=rh.suivante)
12:       if (lh.val != rh.val)
13:        return 0;
14:
15:    return 1;
       // s'ils ne sont pas différents, ils doivent correspondre
16:  }
```

8. Oui, parce que la comparaison des tableaux implique la comparaison de leurs éléments. `operator!=` doit être défini pour les éléments également.

9. Voici une réponse possible :

```
 0:  //Exercice 19.9
 1:  // modèle permuter :
 2:  // doit avoir l'affectation et le constructeur de copie
 3:  // définis pour le Type.
 4:  template <class Type>
 5:  void permuter( Type& lhs, Type& rhs)
 6:  {
 7:     Type temp ( lhs );
 8:     lhs = rhs;
 9:     rhs = temp;
10:  }
```

Chapitre 20

Testez vos connaissances

1. Une exception est un objet qui est créé par l'appel du mot-clé `throw`. Elle signale une situation exceptionnelle et est transmise par la pile des appels vers la première instruction `catch` qui gère son type.
2. Un bloc `try` est un ensemble d'instructions qui peuvent produire une exception.
3. Une instruction `catch` est un traitement qui a une signature du type d'exception qu'elle gère. Elle suit un bloc `try` et reçoit les exceptions déclenchées dans ce bloc.
4. Une exception est un objet pouvant contenir toute information définissable dans une classe créée par l'utilisateur.
5. Les objets exception sont créés avec le mot-clé `throw`.
6. En général, on transmet les exceptions par référence. Si vous n'avez pas l'intention de modifier le contenu de l'objet exception, vous devez passer une référence `const`.
7. Oui, si vous transmettez l'exception par référence.
8. Les instructions `catch` sont examinées selon leur ordre d'apparition dans le code source. C'est la première instruction `catch` dont la signature correspond qui sera utilisée. En général, il vaut mieux commencer avec l'exception la plus spécifique et finir par la plus générale.
9. `catch(...)` intercepte toutes les exceptions quel qu'en soit le type.
10. Un point d'arrêt est un emplacement du code où le débogueur va arrêter l'exécution.

Exercices

1. Voici une réponse possible :

```
 0:  #include <iostream>
 1:  using namespace std;
 2:  class OutOfMemory {};
 3:  int main()
 4:  {
 5:     try
 6:     {
 7:       int *monInt = new int;
 8:       if (monInt == 0)
 9:           throw OutOfMemory();
10:     }
```

```
11:     catch (OutOfMemory)
12:     {
13:        cout << "Impossible d'allouer la mémoire !" << endl;
14:     }
15:  return 0;
16:  }
```

2. Voici une réponse possible :

```
 1:  #include <iostream>
 2:  #include <stdio.h>
 3:  #include <string.h>
 4:  using namespace std;
 5:  class OutOfMemory
 6:  {
 7:    public:
 8:     OutOfMemory(char *);
 9:     char* GetString() { return saString;}
10:  private:
11:     char* saString;
12:  };
13:
14:  OutOfMemory::OutOfMemory(char * leType)
15:  {
16:     saString = new char[80];
17:     char warning[]= "Mémoire saturée ! allocation impossible
17a:                     pour: ";
18:     strncpy(saString, warning, 60);
19:     strncat(saString, leType,19);
20:  }
21:  int main()
22:
23:  {
24:     try
25:     {
26:        int *monInt = new int;
27:        if (monInt == 0)
28:            throw OutOfMemory("int");
29:     }
30:     catch (OutOfMemory& uneException)
31:     {
32:        cout << uneException.GetString();
33:     }
34:     return 0;
35:  }
```

3. Voici une réponse possible :

```
 0:  // Exercice 20.3
 1:  #include <iostream>
 2:  using namespace std;
 3:  // type de données d'exception abstraite
 4:  class Exception
 5:  {
 6:     public:
 7:        Exception(){}
 8:        virtual ~Exception (){}
 9:        virtual void AffErreur() = 0;
10:  };
11:
12:  // Classe dérivée pour gérer les problèmes de mémoire.
13:  // Remarquez l'absence d'allocation mémoire dans cette
13a: // classe !
14:  class OutOfMemory : public Exception
15:  {
16:  public:
17:     OutOfMemory(){}
18:     ~OutOfMemory(){}
19:     virtual void AffErreur();
20:  private:
21:  };
22:
23:  void OutOfMemory::AffErreur()
24:  {
25:     cout << "Problème de mémoire !!" << endl;
26:  }
27:
28:  //classe dérivée pour gérer les nombres erronés
29:  class RangeError : public Exception
30:  {
31:  public:
32:     RangeError(unsigned long nombre) {mauvaisNombre = nombre;}
33:     ~RangeError () {}
34:     virtual void AffErreur();
35:     virtual unsigned long GetNombre() {return mauvaisNombre; }
36:     virtual void SetNombre(unsigned long nombre)
36a:        {mauvaisNombre = nombre;}
37:  private:
38:     unsigned long mauvaisNombre;
39:  };
40:
```

```
41:  void RangeError::AffErreur()
42:  {
43:     cout << "Le nombre "
44:     cout << GetNombre() << "est incorrect!!" << endl;
45:  }
46:
47:  void maFonction(); // prototype de fonc.
48:
49:  int main()
50:  {
51:     try
52:     {
53:     maFonction();
54:     }
55:     // Un seul catch nécessaire, on utilise les fonctions
56:     // virtuelles pour faire ce qu'il faut
57:     catch (Exception& uneException)
58:     {
59:         uneException.AffErreur();
60:     }
61:     return 0;
62:   }
63:
64:  void maFonction()
65:  {
66:     unsigned int *monInt = new unsigned int;
67:     long testNombre;
68:
69:     if(monInt == 0)
70:        throw OutOfMemory();
71:
72:     cout << "Entrez un entier : ";
73:     cin >> testNombre;
74:
75:     // ce test bizarre pourrait être remplacé par une série
76:     // de tests pour signaler une mauvaise saisie de
76a:    // l'utilisateur.
77:
78:     if (testNombre > 3768 || testNombre < 0)
79:      throw RangeError(testNombre);
80:
81:     *monInt = testNombre
82:     cout << "OK, monInt : " << *monInt;
83:     delete monInt;
84:  }
```

4. Voici une réponse possible :

```
 0:  // Exercice 20.4
 1:  #include <iostream>
 2:  using namespace std;
 3:  // Type de données d'exception abstraite
 4:  class Exception
 5:  {
 6:  public:
 7:     Exception(){}
 8:     virtual ~Exception (){}
 9:     virtual void AffErreur() = 0;
10:   };
11:
12:  // classe dérivée pour gérer les problèmes de mémoire.
13:  // Remarquez l'absence d'allocation de mémoire dans cette
13a: // classe !
14:  class OutOfMemory : public Exception
15:  {
16:  public:
17:     OutOfMemory(){}
18:     ~OutOfMemory(){}
19:   virtual void AffErreur();
20:  private:
21:  };
22:
23:  void OutOfMemory::AffErreur()
24:  {
25:     cout << "Mémoire saturée !!\n";
26:  }
27:
28:  //classe dérivée pour gérer les nombres erronés
29:  class RangeError : public Exception
30:  {
31:  public:
32:     RangeError(unsigned long nombre) {mauvaisNombre = nombre;}
33:     ~RangeError () {}
34:     virtual void AffErreur();
35:     virtual unsigned long GetNombre() {return mauvaisNombre; }
36:     virtual void SetNombre(unsigned long nombre)
36a:       {mauvaisNombre = nombre;}
37:  private:
38:     unsigned long mauvaisNombre;
39:  };
40:
41:  void RangeError::AffErreur()
42:  {
```

```
43:     cout << "Le nombre "
44:     Cout << GetNombre() << "est incorrect!!" << endl;
45:  }
46:
47:  // prototype de fonc.
48:  void maFonction();
49:  unsigned int * FonctionDeux();
50:  void FonctionTrois(unsigned int *)
51:
52:  int main()
53:  {
54:     try
55:     {
56:        maFonction();
57:     }
58:   // Un seul catch est nécessaire, on utilise les fonctions
59:   // virtuelles pour faire ce qu'il faut
60:   catch (Exception& uneException)
61:   {
62:       uneException.AffErreur();
63:   }
64:   return 0;
65:   }
66:
67:   unsigned int * FonctionDeux()
68:   {
69:      unsigned int *monInt = new unsigned int;
70:      if (monInt == 0)
71:         throw OutOfMemory();
72:      return monInt;
73:   }
74:
75:   void maFonction()
76:   {
77:      unsigned int *monInt = fonctionDeux();
78:      FonctionTrois(monInt);
79:      cout << "OK. monInt : " << *monInt;
80:      delete monInt;
81:   }
82:
83:   void FonctionTrois(unsigned int *ptr)
84:   {
85:      long testNombre;
86:      cout << "Entrez un int: ";
87:      cin >> testNombre;
88:      // ce test bizarre pourrait être remplacé par une série
89:      // de tests pour signaler une mauvaise saisie de
```

```
89a:     // l'utilisateur.
90:      if (testNombre > 3768 || testNombre < 0)
91:         throw RangeError(testNombre);
92:      *ptr = testNombre;
93:   }
```

5. Dans le processus de gestion d'un manque de mémoire, le constructeur de `xOutOfMemory` crée un objet chaîne. Cette exception ne peut être lancée que lorsque le programme manque de mémoire et cette l'allocation va donc échouer.

   Il est possible que la tentative de création de cette chaîne déclenche la même exception, ce qui va créer une boucle infinie jusqu'à ce que le programme s'arrête. Si cette chaîne est réellement nécessaire, vous pouvez allouer l'espace dans un tampon statique avant de commencer le programme, puis l'utiliser si nécessaire au moment de l'envoi de l'exception.

   Vous pouvez tester ce programme en remplaçant la ligne `if (var == 0)` par `if (1)`, qui force le déclenchement de l'exception.

Chapitre 21

Testez vos connaissances

1. `#ifndef` permet d'éviter l'inclusion multiple d'un fichier en-tête dans un programme.
2. La réponse à cette question dépend du compilateur que vous utilisez.
3. `#define debug 0` définit le terme `debug` comme étant la valeur 0 (zéro). Pour chaque mot `debug` trouvé, le caractère `0` sera donc substitué. `#undef debug` supprime toutes les définitions de `debug`. Tous les mots `debug` du fichier ne seront pas substitués.
4. La réponse est `4 / 2`, c'est-à-dire `2`.
5. Le résultat est `10 + 10 / 2`, c'est-à-dire `10 + 5`, soit `15`. Ce n'est évidemment pas le résultat souhaité.
6. Il faut ajouter les parenthèses :

   ```
   MOITIE(x) ((x)/2)
   ```

7. Deux octets font 16 bits, on peut donc stocker jusqu'à 16 bits.
8. Cinq bits peuvent contenir 32 valeurs (de 0 à 31).
9. Le résultat est 1111 1111.
10. Le résultat est 0011 1100.

Exercices

1. Voici la réponse :

```
#ifndef STRING_H
#define STRING_H
...
#endif
```

2. Voici une réponse possible :

```
 0:  #include <iostream>
 1:
 2:  using namespace std;
 3:  #ifndef DEBUG
 4:  #define ASSERT(x)
 5:  #elif DEBUG == 1
 6:  #define ASSERT(x) \
 7:     if (! (x)) \
 8:     {  \
 9:       cout << "ERREUR !! Assert " << #x << " a échoué\n"; \
10:     }
11:  #elif DEBUG == 2
12:  #define ASSERT(x)  \
13:     if (! (x) ) \
14:     {  \
15:       cout << "ERREUR !! Assert " << #x << " a échoué\n";  \
16:       cout << " à la ligne " << __LINE__ << endl;  \
17:       cout << " du fichier " << __FILE__ << endl;  \
18:     }
19:  #endif
```

3. Voici une macro possible :

```
#ifndef DEBUG
#define DPRINT(string)
#else
#define DPRINT(STRING) cout << #STRING;
#endif
```

4. Voici une réponse possible :

```
class maDate
{
  public:
    // Éléments ici...
  private:
    unsigned int Mois : 4;
    unsigned int Jour   : 8;
    unsigned int Annee  : 12;
}
```



D

Étude des listes chaînées

Le Chapitre 13 vous a fait découvrir les tableaux et les listes chaînées. Une liste chaînée est une structure de données constituée de petits conteneurs conçus pour être liés ensemble, en fonction des besoins. Le principe consiste à écrire une classe contenant un seul objet de données et qui peut pointer sur le prochain conteneur du même type. On crée ensuite un conteneur pour chaque objet à stocker et on les chaîne ensemble, en fonction des besoins.

Les conteneurs sont appelés nœuds. Le premier de la liste s'appelle la tête, et le dernier la queue.

Il existe trois types de listes de base qui sont, de la plus simple à la plus complexe :

- les listes simplement chaînées ;
- les listes doublement chaînées ;
- les arborescences.

Dans une liste simplement chaînée, chaque nœud pointe vers le suivant, jamais vers l'arrière. Pour retrouver un nœud, on commence donc au début et l'on passe de nœud en nœud. Une liste doublement chaînée permet d'avancer ou de reculer dans la chaîne.

Quant à l'arborescence, il s'agit d'une structure complexe construite à partir de nœuds, chacun pointant dans deux directions ou plus. La Figure D.1 présente ces trois structures.

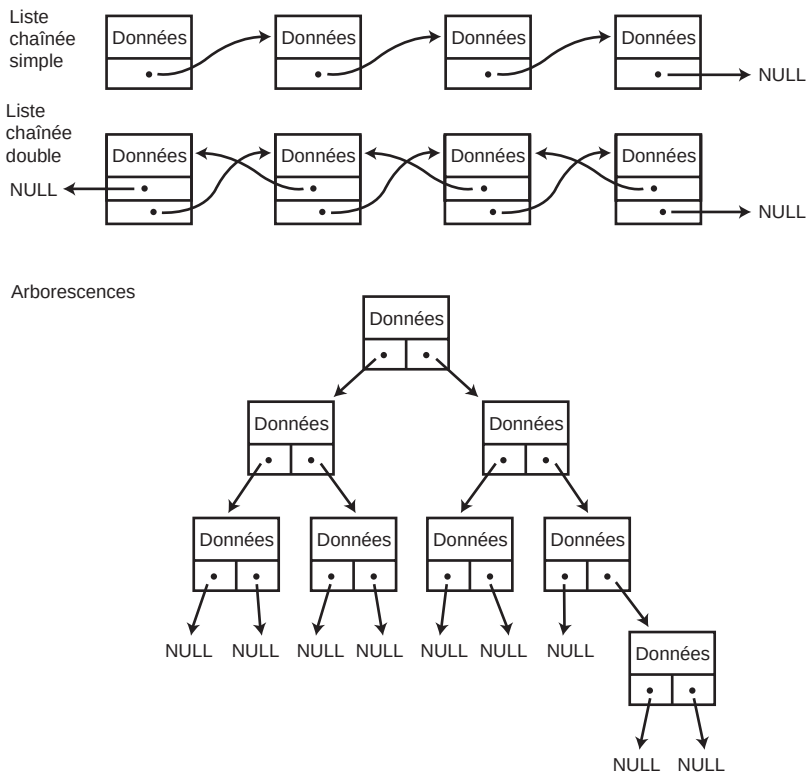


Figure D.1
Des listes chaînées.

Dans cette annexe, nous étudierons en détail une liste chaînée pour comprendre comment créer des structures complexes et, ce qui est plus important, comment les utiliser.

Les composants d'une liste chaînée

Les listes chaînées sont composées de nœuds. La classe du nœud sera elle-même abstraite. Vous utiliserez trois sous-types pour accomplir cette tâche : un nœud de tête – chargé de gérer la tête de la liste –, un nœud de queue et zéro nœud interne ou plus. Les nœuds internes assureront le suivi des données contenues dans la liste.

Notez que les données et la liste sont deux choses différentes. En théorie, vous pouvez enregistrer n'importe quel type de données dans une liste : ce ne seront pas elles qui seront reliées, mais les nœuds qui *contiennent* les données.

Le programme utilisateur ne sait rien des nœuds : il manipule une liste qui, elle, fait peu de choses puisqu'elle se contente de déléguer le travail aux nœuds.

Le Listing D.1 présente un code que nous étudierons en détail dans le reste de cette annexe.

Listing D.1 : Une liste chaînée

```
// ********************************************
//    FICHIER :   Listing D.1
//    OBJET :     Présenter une liste chaînée
//    NOTES :
//
//  COPYRIGHT : Copyright (C) 2000-04 Liberty Associates, Inc.
//                Tous droits réservés
//
// Présente une approche orientée objet des
// listes chaînées. La liste délègue au noeud.
// Le noeud est un type de données abstrait. Il existe trois
// types de noeuds : de tête, de queue et internes.
// Seuls les noeuds internes contiennent des données.
//
// La classe Donnees sert à contenir les objets
// de la liste chaînée.
//
// ********************************************


#include <iostream>
using namespace std;

enum { kPlusPetit, kPlusGrand, kIdentique};

// Classe Donnees pour insérer dans la liste chaînée
// Toute classe de cette liste doit accepter deux méthodes :
// Affiche (affiche la valeur) et
// Compare (renvoie la position relative)
class Donnees
{
  public:
    Donnees(int val): maValeur(val){}
    ~Donnees(){}
    int Compare(const Donnees &);
    void Affiche() { cout << maValeur << endl; }
  private:
    int maValeur;
};
```

```
 40:
 41:  // Compare permet de décider de l'endroit dans la liste
 42:  // où placer un objet particulier.
 43:  int Donnees::Compare(const Donnees & AutresDonnees)
 44:  {
 45:     if (maValeur < AutresDonnees.maValeur)
 46:        return kPlusPetit;
 47:     if (maValeur > AutresDonnees.maValeur)
 48:        return kPlusGrand;
 49:     else
 50:        return kIdentique;
 51:  }
 52:
 53:  // Déclarations anticipées
 54:  class Noeud;
 55:  class NoeudTete;
 56:  class NoeudQueue;
 57:  class NoeudInterne;
 58:
 59:  // TDA représentant l'objet noeud dans la liste
 60:  // Chaque classe dérivée doit surcharger Insere et Affiche
 61:  class Noeud
 62:  {
 63:    public:
 64:      Noeud(){}
 65:      virtual ~Noeud(){}
 66:      virtual Noeud * Insere(Donnees * lesDonnees)=0;
 67:      virtual void Affiche() = 0;
 68:    private:
 69:  };
 70:
 71:  // Voici le noeud contenant l'objet réel
 72:  // Ici, l'objet est de type Donnees
 73:  // Nous verrons comment le rendre plus général
 74:  // lorsque nous traiterons des modèles
 75:  class NoeudInterne: public Noeud
 76:  {
 77:    public:
 78:      NoeudInterne(Donnees * lesDonnees, Noeud * suivant);
 79:      ~NoeudInterne(){ delete leSuivant; delete mesDonnees; }
 80:      virtual Noeud * Insere(Donnees * lesDonnees);
 81:      // Déléguer !
 82:      virtual void Affiche() { mesDonnees->Affiche();
82a:                               leSuivant->Affiche(); }
 83:
 84:    private:
 85:      Donnees * mesDonnees;  // Les données
```

```
 86:      Noeud * leSuivant;    // Pointe vers le prochain noeud
 87:  };
 88:
 89:  // Le constructeur se contente d'initialiser
 90:  NoeudInterne::NoeudInterne(Donnees * lesDonnees, Noeud * suivant):
 91:    mesDonnees(lesDonnees),leSuivant(suivant)
 92:  {
 93:  }
 94:
 95:  // L'essentiel de la liste
 96:  // Lorsque vous placez un nouvel objet dans la liste
 97:  // il est passé au noeud, qui décide
 98:  // où il va et l'insère dans la liste
 99:  Noeud * NoeudInterne::Insere(Donnees * lesDonnees)
100:  {
101:
102:     // Le nouveau est-il plus grand ou plus petit que moi ?
103:     int resultat = mesDonnees->Compare(*lesDonnees);
104:
105:
106:     switch(resultat)
107:     {
108:     // Par convention, s'il est pareil que moi, il passe en 1er
109:     case kIdentique:
110:     case kPlusGrand:    // Les nouvelles données viennent avant moi
111:        {
112:           NoeudInterne * donneesNoeud =
112a:                 new NoeudInterne(lesDonnees, this);
113:           return donneesNoeud;
114:        }
115:
116:     // Il est plus grand que moi, le passer au noeud
117:     // suivant et le laisser gérer.
118:     case kPlusPetit:
119:        leSuivant = leSuivant->Insere(lesDonnees);
120:        return this;
121:     }
122:     return this;
123:  }
124:
125:
126:  // Le noeud de queue n'est qu'une sentinelle
127:
128:  class NoeudQueue : public Noeud
129:  {
130:    public:
131:      NoeudQueue(){}
```

```
 132:      ~NoeudQueue(){}
 133:      virtual Noeud * Insere(Donnees * lesDonnees);
 134:      virtual void Affiche() { }
 135:
 136:    private:
 137:
 138:  };
 139:
 140:  // Si les données viennent à moi, elles doivent être insérées
 141:  // avant moi car je suis la queue et RIEN ne vient après
 142:  Noeud * NoeudQueue::Insere(Donnees * lesDonnees)
 143:  {
 144:     NoeudInterne * donneesNoeud =
144a:            new NoeudInterne(lesDonnees, this);
 145:     return donneesNoeud;
 146:  }
 147:
 148:  // Le noeud de tête n'a pas de données, il se contente de pointer
 149:  // vers le début de la liste
 150:  class NoeudTete : public Noeud
 151:  {
 152:    public:
 153:      NoeudTete();
 154:      ~NoeudTete() { delete leSuivant; }
 155:      virtual Noeud * Insere(Donnees * lesDonnees);
 156:      virtual void Affiche() { leSuivant->Affiche(); }
 157:    private:
 158:      Noeud * leSuivant;
 159:  };
 160:
 161:  // Dès sa création, la tête
 162:  // crée la queue
 163:  NoeudTete::NoeudTete()
 164:  {
 165:     leSuivant = new NoeudQueue;
 166:  }
 167:
 168:  // Rien ne vient avant la tête,
 169:  // donc passer les données au noeud suivant
 170:  Noeud * NoeudTete::Insere(Donnees * lesDonnees)
 171:  {
 172:     leSuivant = leSuivant->Insere(lesDonnees);
 173:     return this;
 174:  }
 175:
 176:  // J'ai tout le mérite et ne fais rien
 177:  class ListeChainee
```

```
178: {
179:   public:
180:     ListeChainee();
181:     ~ListeChainee() { delete maTete; }
182:     void Insere(Donnees * lesDonnees);
183:     void AfficheTout() { maTete->Affiche(); }
184:   private:
185:     NoeudTete * maTete;
186: };
187:
188: // À la naissance, je crée le noeud de tête
189: // Celui-ci crée le noeud de queue
190: // Une liste vide pointe donc vers la tête qui
191: // pointe vers la queue et ne contient rien
192: ListeChainee::ListeChainee()
193: {
194:    maTete = new NoeudTete;
195: }
196:
197: // Déléguer, déléguer, déléguer
198: void ListeChainee::Insere(Donnees * pDonnees)
199: {
200:    maTete->Insere(pDonnees);
201: }
202:
203: // Programme de test
204: int main()
205: {
206:    Donnees * pDonnees;
207:    int val;
208:    ListeChainee ll;
209:
210:    // Demander à l'utilisateur de produire des valeurs
211:    // les placer dans la liste
212:    for (;;)
213:    {
214:       cout << "Insérer une valeur (0 pour arrêter) : ";
215:       cin >> val;
216:       if (val == 0)
217:          break;
218:       pDonnees = new Donnees(val);
219:       ll.Insere(pDonnees);
220:    }
221:
222:    // Parcourir la liste et afficher les données
223:    ll.AfficheTout();
224:    return 0;  // ll est hors de portée, donc détruit !
225: }
```

Ce programme produit le résultat suivant :

```
Insérer une valeur (0 pour arrêter) : 5
Insérer une valeur (0 pour arrêter) : 8
Insérer une valeur (0 pour arrêter) : 3
Insérer une valeur (0 pour arrêter) : 9
Insérer une valeur (0 pour arrêter) : 2
Insérer une valeur (0 pour arrêter) : 10
Insérer une valeur (0 pour arrêter) : 0
2
3
5
8
9
10
```

La première chose à remarquer concerne la constante énumérée définie à la ligne 24, qui fournit trois valeurs constantes : `kPlusPetit`, `kPlusGrand` et `kIdentique`. Chaque objet qui pourrait être contenu dans cette liste chaînée doit fournir une méthode `Compare()`. Ces constantes seront le résultat renvoyée par cette méthode `Compare()`.

Pour les besoins de l'exemple, la classe `Donnees` est créée aux lignes 30 à 39. La méthode `Compare()` est implémentée aux lignes 43 à 51. Un objet `Donnees` contient une valeur et peut se comparer à d'autres objets `Donnees`. Il fournit également une méthode `Affiche()` qui affiche la valeur de l'objet `Donnees`.

Pour comprendre le fonctionnement de la liste chaînée, le plus simple est d'étudier un exemple pas à pas. La ligne 203 commence le programme de test ; la ligne 206 déclare un pointeur vers un objet `Donnees`. Enfin, la ligne 208 définit une liste chaînée locale.

Étudiez la classe `ListeChainee` aux lignes 177 à 186. Lorsqu'elle est créée, le constructeur de la ligne 192 est appelé. La seule tâche effectuée dans le constructeur consiste à allouer un objet `NoeudTete` et à affecter l'adresse de cet objet au pointeur de tête de la liste chaînée, qui est déclaré à la ligne 185.

Cette affectation appelle le constructeur `NoeudTete` implémenté aux lignes 163 à 166. À son tour, celui-ci alloue un `NoeudQueue` et affecte son adresse au pointeur `leSuivant` du nœud de tête. La création du nœud de queue appelle le constructeur `NoeudQueue` présenté à la ligne 131, qui est déclaré *inline* et qui ne fait rien.

Ainsi, le simple fait d'allouer une liste chaînée sur la pile crée cette liste, un nœud de tête et un nœud de queue, ainsi que les liaisons entre eux, comme le montre la Figure D.2.

La ligne 212 du programme principal commence une boucle sans fin. L'utilisateur doit alors entrer des valeurs qui seront ajoutées à la liste chaînée. Il peut ajouter autant de valeurs qu'il le souhaite, puis `0` lorsqu'il a terminé. Le code de la ligne 216 évalue la valeur saisie : si elle est égale à `0`, le programme interrompt la boucle.

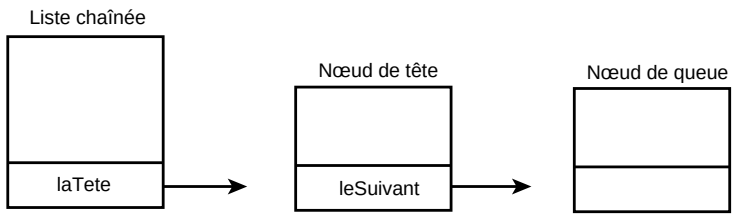


Figure D.2
La liste chaînée après sa création.

Dans le cas contraire, la ligne 218 crée un objet `Donnees`, qui est inséré dans la liste à la ligne 219. Si l'utilisateur a saisi par exemple la valeur `15`, la méthode `Insere` est appelée à la ligne 198.

La liste chaînée délègue immédiatement la responsabilité de l'insertion de l'objet à son nœud de tête, ce qui provoque l'appel de la méthode `Insere` de la ligne 170. Le nœud de tête passe immédiatement la responsabilité au nœud vers lequel pointe `leSuivant`. Dans ce (premier) cas, il pointe vers le nœud de queue (n'oubliez pas que, lors de sa création, le nœud de tête a créé un lien vers le nœud de queue). Cela appelle donc la méthode `Insere` de la ligne 142.

`NoeudQueue::Insere` sait que l'objet qu'il gère doit être inséré immédiatement avant lui, c'est-à-dire que le nouvel objet se trouvera dans la liste juste avant le nœud de queue. Ainsi, à la ligne 144, il crée un nouvel objet `NoeudInterne`, en lui passant les données et un pointeur vers lui-même, ce qui provoque l'appel du constructeur de l'objet `NoeudInterne` de la ligne 90.

Le constructeur `NoeudInterne` ne fait rien d'autre que d'initialiser son pointeur `Donnees` avec l'adresse de l'objet `Donnees` qu'il a reçu, puis son pointeur `leSuivant` avec l'adresse du nœud qu'il a reçu. Dans ce cas, le nœud vers lequel il pointe est le nœud de queue (n'oubliez pas que le nœud de queue lui a passé son propre pointeur `this`).

Maintenant que `NoeudInterne` a été créé, son adresse est affectée au pointeur `donnees-Noeud` ligne 144, puis renvoyée à partir de la méthode `NoeudQueue::Insere()`. Cela nous renvoie à `NoeudTete::Insere()`, où l'adresse du `NoeudInterne` est affectée au pointeur `leSuivant` de `NoeudTete` (ligne 172). Enfin, l'adresse de `NoeudTete` est renvoyée à la liste chaînée où, ligne 200, elle est supprimée (on n'en fait rien, car la liste chaînée connaît déjà l'adresse du nœud de tête).

Pourquoi s'inquiéter de renvoyer l'adresse si elle n'est pas utilisée ? `Insere` est déclarée dans la classe de base, `Noeud`. Or, la valeur renvoyée est nécessaire aux autres implémentations. Si vous modifiez la valeur de retour de `NoeudTete::Insere()`, vous recevrez une erreur du compilateur ; il est donc plus simple de renvoyer le `NoeudTete` et de laisser la liste chaînée se débarrasser de son adresse.

Que s'est-il donc passé ? Les données ont été insérées dans la liste. La liste les a passées à sa tête. La tête les a passées à l'élément vers lequel elle pointe. Dans ce (premier) cas, la tête pointait vers la queue. La queue a immédiatement créé un nouveau nœud interne, initialisant ainsi le nouveau nœud pour qu'il pointe vers la queue. La queue a renvoyé l'adresse du nouveau nœud à la tête, qui a réaffecté son pointeur `leSuivant` de sorte qu'il pointe vers le nouveau nœud. Et voilà ! les données se trouvent dans la liste, au bon endroit, comme l'indique la Figure D.3.

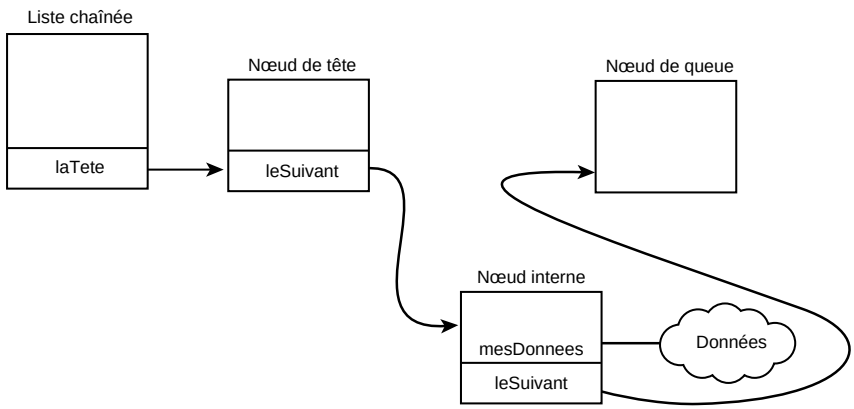


Figure D.3
La liste chaînée après insertion du premier nœud.

Après l'insertion du premier nœud, le contrôle du programme reprend à la ligne 214. Une fois de plus, on évalue la valeur qui a été saisie. Pour notre exemple, supposons que nous entrions la valeur `3`. Un nouvel objet `Donnees` est alors créé à la ligne 218, pour être inséré dans la liste à la ligne 219.

À nouveau, à la ligne 200 la liste passe les données à son `NoeudTete`. La méthode `NoeudTete::Insere()` passe à son tour la nouvelle valeur à l'élément vers lequel pointe `leSuivant`. Comme nous venons de le voir, celui-ci pointe maintenant vers le nœud qui contient l'objet `Donnees` valant `15`. Cela provoque l'appel de la méthode `NoeudInterne::Insere()` de la ligne 99.

À la ligne 103, le `NoeudInterne` utilise son pointeur `lesDonnees` pour indiquer à son objet `Donnees` (qui vaut `15`) d'appeler sa méthode `Compare()` en passant le nouvel objet `Donnees` (qui vaut `3`), ce qui provoque l'appel de la méthode `Compare()` de la ligne 43.

Les deux valeurs sont comparées. Comme `maValeur` vaudra 15 et que `AutresDonnees.maValeur` vaudra 3, la valeur renvoyée sera `kPlusGrand`. Le flux du programme passe donc au cas `kPlusGrand` ligne 110.

Un nouveau `NoeudInterne` est créé pour le nouvel objet `Donnees`. Le nouveau nœud pointe vers l'objet `NoeudInterne` courant et l'adresse du nouveau `NoeudInterne` est renvoyée à partir de la méthode `NoeudInterne::Insere()` vers le `NoeudTete`.

Le nouveau nœud, dont la valeur d'objet est inférieure à la valeur de l'objet du nœud courant, est donc inséré dans la liste qui ressemble maintenant à la Figure D.4.

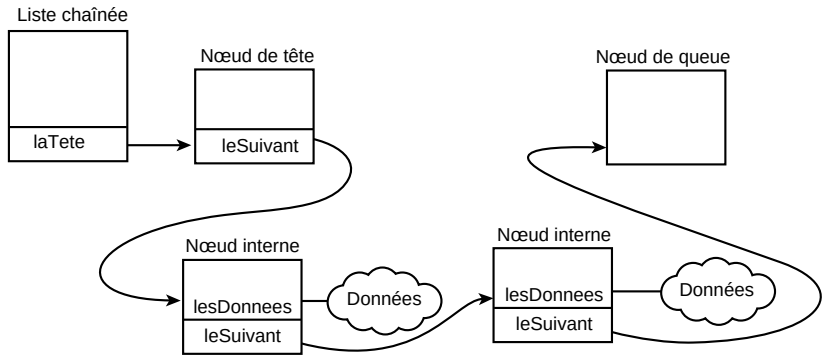


Figure D.4
La liste chaînée après insertion du second nœud.

Dans le troisième appel de la boucle, le client ajoute la valeur `8`. Celle-ci est supérieure à 3, mais inférieure à 15 et doit donc être insérée entre les deux nœuds existants. La progression est donc identique à l'exemple précédent, sauf que lorsque le nœud dont la valeur d'objet est 3 effectue la comparaison, il renvoie `kPlusPetit` au lieu de `kPlusGrand` – ce qui signifie que l'objet valant 3 est plus petit que le nouvel objet valant 8. La méthode `NoeudInterne::Insere()` va donc effectuer un branchement sur le cas `kPlusPetit` de la ligne 118. Au lieu de créer un nouveau nœud et de l'insérer, le `NoeudInterne` passe simplement les nouvelles données à la méthode `Insere` de l'élément vers lequel pointe `leSuivant`. Dans ce cas, il appelle donc `InsereNoeud` sur le `NoeudInterne` dont la valeur de l'objet `Donnees` est 15.

La comparaison est à nouveau réalisée et un nouvel objet `NoeudInterne` est créé. Ce dernier pointe vers le `NoeudInterne` dont l'objet `Donnees` vaut 15 et son adresse est repassée au `NoeudInterne` dont l'objet `Donnees` vaut 3, comme le montre la ligne 119.

De ce fait, le nouveau nœud est inséré dans la liste, au bon endroit.

Si possible, essayez de suivre l'insertion de plusieurs nœuds à partir de votre débogueur. Vous devriez pouvoir étudier l'appel de ces méthodes entre elles et l'ajustement correct des pointeurs.

Qu'avez-vous appris ?

Dans un programme orienté objet correctement conçu, *personne* n'est responsable. Chaque objet fait un petit travail, avec pour résultat une machine bien huilée.

La liste chaînée a pour seule tâche de maintenir le nœud de tête, qui passe immédiatement les nouvelles données à l'élément vers lequel il pointe, sans s'occuper de qui il est.

Le nœud de queue crée un nouveau nœud et l'insère dès qu'il reçoit des données. Il ne sait qu'une seule chose : si un élément vient à lui, celui-ci sera inséré juste avant lui.

Les nœuds internes sont un peu plus complexes ; ils demandent à leur objet existant de se comparer avec le nouvel objet. Selon le résultat, ils l'insèrent ou le font passer.

Vous remarquerez que le nœud interne (`NoeudInterne` dans ce listing) ne *sait* pas comment effectuer la comparaison ; c'est de la responsabilité de l'objet lui-même. Tout ce qu'il sait faire est de demander aux objets de se comparer et d'attendre l'une des trois réponses possibles. S'il reçoit une réponse, il insère l'élément, sinon il le transmet sans savoir ni se soucier de l'endroit où il atterrira.

Index

Symboles

738, 745
#include (instruction) 24
// 17, 26, 580
>> 580, 589
& 210, 244, 759
&& 759
* 214
++ 53
= 48
|| 759

A

Accès 769
  agrégés 523
  d'une classe 141
  données et fonctions membres 228
  membre de classe 228
Accolades 25, 765
  dans classe 139
  dans définition de fonction 161
  dans énumération 59
  dans fonction 34, 102
Acteur 328
Activation de bits 760
Adressage indirect 213
  opérateur 214
Adresse
  mémoire variable 210
  mémoire *Voir* Opérateurs
  pointeur 215
  stockage dans un pointeur 212
Affectation 66
  de valeurs à une variable 141
  opérateur d' *Voir* Opérateurs
Affichage d'une chaîne 566
Agrégation 523
  coût 524
  présentation 515
Algorithmes 693
  Voir aussi Conteneurs
Alias 50, 244
  d'espace de noms 634
Alignement de caractères en sortie 28
Amies
  classes *Voir* Classes
  fonctions *Voir* Fonctions
Analyse 13, 338
  des besoins 327
  documents d' 339
  et conception orientée objet 322
Anonyme, variable temporaire 299
ANSI 12
Appel de fonction 98, 128
argc (argument) 613, 614
Arguments
  de fonction 111
  par valeur 107

argv (argument) 613
ASCII 14, 43, 54
Associatifs, conteneurs *Voir* Conteneurs
Association (POO) 334
Astérisque (opérateur) 214
Avertissements du compilateur 21

B

Bibliothèque 15
  définition 576
  iostream 575
  modèles standard 641, 676
  standard 18, 25, 28
    Voir aussi std
Bits, champs 761
Blocs
  catch 703, 706
  multiples 713
  try 706, 712
bool 44
Booléen 77
Boucles 173
  complexes 177
  for 186
  instructions nulles 189
  sans fin 201
  while (true) 181
break, syntaxe 181

C

C# 11
C++
  et langage C 11
  et programmation orientée objet 9
  évolution 11
  historique 5
  liste des mots-clés 775
  notions 5
Caractères
  alignement 28
  ASCII *Voir* ASCII
  blanc 66
  casse 46, 768
  chaîne de 26, 582, 745
  cout 603
  d'échappement 56
  de formatage 26, 28
  fin de ligne (EOL) *Voir* EOL
  spéciaux 56
Cas d'utilisation 327
  paquetages 338
  représentation 336
catch (bloc) 704
cerr 579
Chaînage ascendant 457
Chaîne
  de caractères 26
  type C 421
char 44, 54
cin 25, 36, 579
cin, valeur 585
cin.ignore() (fonction) 592
class (mot-clé) 139, 644
Classes
  accès 362
  conteneurs 677
  d'algorithmes 692
  de conception 341
  déclaration 139
  dérivées 349, 358, 549
  encapsulation 138
  exception 719
  fstream 579
  interface 156
  ios 579
  iostream 579
  istream 579
  membres de 138
  modèles 652
  streambuf 578
  string 425
  surcharge de constructeurs 367
  tableaux 434
Clés 691
Client 156, 160
clog 579
Clonage d'objet 388
Code
  accolade *Voir* Accolades
  assembleur 733
  commentaires 30, 768
  compilation 6
  espace de 129
  fractionnement 42, 766
  lisibilité 766
  source *Voir* Fichiers sources
Commentaires dans le code 30
Compilation 6
  du code 6
  erreurs 15, 20, 143, 147
  messages d'avertissement 21
  préprocesseur 24, 738
Composition 333
  ou héritage multiple 348
Concaténation 746
Conception 341
  application 13
Conflit de noms, résoudre 619
const 770
  fonctions membres 237
  pointeur 237

const this (pointeur) 240
Constantes 57
  définition avec #define 58
  définition avec const 59
  énumérées 59
  littérales 57
  symboliques 58
Constructeurs 365
  ajout 151
  objets de l'héritage multiple 449
  par défaut 152, 159, 285
  privés 432, 562
  surcharge 285
  virtuels 388
Construction
  d'un projet 19
  de données membres 229
Contacter l'auteur 771
Conteneurs 676
  associatifs 687
  avec modification 694
  deque 687
  list 684
  map 688
  multimap 691
  multiset 692
  queue 687
  sans modification 693
  séquentiels 677
  set 691
  stack 686
  vector 677
continue, syntaxe 181
Conventions
  d'attribution de nom 139
  du livre 2
Conversion
  de type *Voir* Transtypage
  implicite et explicite 316
  opérateurs de 316
  printf() (spécificateurs) 601
Copie
  constructeur de 288
  de surface 289
  en profondeur 289
  locale 261
  par valeur 527
Corps
  boucle 176, 183, 791
  constructeur 287
  espace de noms 620
  fonction 34
cout 25, 29, 36, 579
  présentation 26
  utilisation des opérateurs 126
CPU *Voir* UC
Création
  d'un espace de noms 625
  d'une variable 44
  de nouveaux types 137
  de plusieurs variables 48

D

Débogueur
  mode DEBUG 755
  Voir aussi Point de rupture
DEBUG 748
Déclaration
  d'héritage multiple 448
  de classes 139, 159
  de fonctions 99
  de membres de classe 142
  de pointeurs 220
  de tableaux 402, 404, 414
  et définition, différence 168
  exemple 162
  using 632
  virtuelles 461
default, instruction (switch) 198
Définition
  d'objet 140
  de classe 770
  de fonction 101
  de modèle 643
  de type 50
  des indicateurs 598
Délégation 531
  de propriétés 532
Dépassement, valeur limite des entiers 52
Déréférencement 214
Dérivation
  objet 363
  TAD 473
Désactivation de bits 760
Destructeurs
  ajout 151
  d'objets 226
  par défaut 152
Développement
  cycle 15
  itératif 324
Directives
  #define 742
  #include 24, 100, 160
  using 630
Discrimination 349
do...while
  boucle 184
  syntaxe 185
Domaine, modèle 331
Données
  accès 228
  découpage 385
  gestion avec pointeurs 216

membres 138
privées, ou protégées 362
stockage en mémoire 40
type abstrait 467
double (type de variable) 44

E

Éditeur
de liens 6, 15
de texte 14
et traitement de texte 21
Éléments de tableaux 396
En ligne, fonction 757
Encapsulation 9, 138, 576
End Of File, fin de fichier *Voir* EOF
End Of Line, fin de ligne *Voir* EOL
En-tête
de boucle 193
de fonction 34, 256
fichier d' 24, 741
Entiers, signés 53
Entrée
avec cin 580
de l'utilisateur 36
standard, récupération de chaînes 589
Entrées/sorties, fichiers 604
Énumération 59
Environnement de développement 13
EOF 586, 587, 592, 819
EOL 592
Erreurs
compilation 15, 20, 101
édition de liens 620
limite de tableaux 401
logiques 700
points d'arrêt 732
Espace
ajouter membres 627
anonyme 634
de nom 619
standard 26, 28
std *Voir* std
fonctions 626
nom 625
standard 636
Étapes d'exécution d'un programme 19
État des E/S 605
Évaluation des expressions 68
Exceptions 701
créer une classe 708
détection 712
modèles 727
multiples 713
new 223
produire 706
traitement 704, 708
Exécution
d'un programme, étapes 19
de fonction 103
Exemples
#define 740
accès aux membres public d'une classe 144
adresses d'une référence 245
ajout de données en fin de fichier 607
break et continue 179
chaînes de caractères et cin 581, 582
champs de bits 762
classes
abstraites 463
agrégées 524
amies 551
complète 164
contenant des objets classe String 426, 520
de base communes 454
constructeurs 153, 286, 365, 449
dans classes dérivées 367
de copie 290
virtuels de copie 388
consultation d'une liste avec un itérateur 685
conteneur map 688
conversion 316
de type 314
corps d'une boucle for vide 192
d'affectation 311
délégation à une liste agrégée 532
destructeurs 153, 365
do...while 184
échange d'objets modèles avec les fonctions 660
écriture d'un objet dans un fichier 610
en ligne 122
entrées multiples 583
erreurs de compilation 157
exceptions
avec les modèles 727
multiples 713
fonction
amie non modèle 652
de classe 148
en ligne plutôt qu'une macro 757
et données membres statiques avec modèles 672
membres 280
virtuelles 379, 380

for 187
  imbriquées 193
  vide 190
fuites mémoire 273
get()
  avec paramètres 588
  avec un tableau de caractères 589
  sans paramètres 586
héritage privé 542
hiérarchies de classes et exceptions 717
ignore() 592
inclusion de fichiers en-tête 162
initialisation de variables membres 288
itération ou récursivité 195
lire le contenu d'un pointeur 218
macro assert() simple 747
manipulation de données par pointeurs 216
masquage de méthodes 374
méthode de base depuis méthode redéfinie 377
méthodes
  d'accès 146
  virtuelles pures 469
modèle Array 647
modification de la largeur de la sortie 597
multiple 445
nom de fichier en paramètre de la ligne de commande 614
nulles dans boucle for 190
objet fonction 692
objet temporaire anonyme 299
opérateur
  ++ 296
  ami 561
  d'addition 304
  sur ostream 657
operator 566
ouverture de fichiers en lecture et écriture 606
paramètres de la ligne de commande 613
parenthèses dans les macros 744
passage
  de références à des objets 267
  par référence et fonctions virtuelles dans exceptions 723
  par valeur et découpage de données 385
plusieurs valeurs
  par pointeurs 257
  par référence 259
pointeur
  comme données membres d'une classe 230
  comme paramètres 502
  const 264
  perdu 234
  sur fonctions membres 508
  this 232, 300
polymorphe 119
préfixe et suffixe 303
printf() 602
réaffectation d'une référence 247
récupération de données dans un objet exception 720
récursive 125
redéfinition d'une méthode de base dans une classe dérivée 372
référence
  à un objet inexistant 271
  aux objets 249
sans fin 201
séquentiel avec modification 694
setf 599
simple 360
strcpy() 424
String 516
strncpy() 424
suppression
  d'objets 226
  d'un pointeur 224
sur objets constants 238
swap() avec des références 255
switch 199
tableau
  à plusieurs dimensions 407
  constantes et énumérations dans 403
  d'entiers 396
  d'objets 404
  de pointeurs sur fonctions membres 510
  écriture hors limite 398
  remplissage 422
  stockage dans le tas 409
  sur le tas avec new 416
transtypage descendant 442
typedef pour les pointeurs 505
types abstraits de données 467

utilisation de références 244
valeurs par défaut 282
des paramètres 116
variables locales et globales 109
vecteur et accès à ses éléments 679
virtuel 458
while 175, 183
modifiée 186
true 182
write() 596
Experts du domaine 328
Expressions
C++ 111
décomposition 76
Extensions de fichiers 160
sources 14

F

Fichiers
binaires 610
en-tête 24, 160
iostream.h 24
entrées/sorties 604
exécutables, création 15
include 770
iostream 27
objet 15
sources, extensions 14
FIFO 687
fill() (fonction) 694
find() 693
float (type de variable) 44
flush() (fonction) 595
Flux
de sortie 597
et tampons 578
présentation 575
printf() 601
Fonctionnalités *Voir* Classes
Fonctionnement des exceptions 712
Fonctions 8, 33
accolades 34
arguments 34
bad() 605
cin 585
close() 605
comme argument 111
comportement 128
const 156
de classes 148
de pointeurs 502
déclaration 565
définition 161
en ligne 121
en-tête 34
eof() 605
et pile 131
exemple 35, 102
fail() 605
fill() 598
getline() 590
good() 605
ignore() 592
macro 742
main() 25, 98
membres 138
modèles 651, 683
nature et rôle 7
noms 101
par défaut 115
paramètres 34
parenthèses 743
polymorphes 118
prédéfinies 100
printf() 601
privées 143
pures 467
put() 595
putback() 593
read() 610
résultats multiples 257
return 35
rôle 98
setf() 598
sizeof() 43
spécialisées 665
surchargées 279
taille 111
typedef 504
valeurs par défaut 284
virtuelles 383
width() 597
write() 595
for
utilisation avancée 188
vides 191
for_each() 693
Format ASCII *Voir* ASCII
Formatage 28
données en sortie 594
Voir aussi Caractères
friend (mot-clé) *Voir* Fonctions/Classes
fstream, fichier 605
Fuite mémoire 223, 225

G

Garbage 130
Généralisation (POO) 322, 333
get() (fonction) 586
Globales
fonctions 98

H

Heap (mémoire) *Voir* Tas
Héritage 10
  avancé 515
  classe de base 453
  constructeurs d'objets 449
  déclaration 448
  dérivation 358
  multiple 347, 445
  parties d'un objet 448
  privé 541
  problèmes 462
  public 347
  représentation 323
  simple 437
Hiérarchie
  complexe d'abstraction 472
  exceptions 716

I

If, instruction et erreurs classiques 81
ifstream
  objet 604
Imbrication
  boucles 192
  classes *Voir* Composition
  espaces de noms 627
Implémentation
  en ligne 161
  en termes de 532
Impression 26
  et caractères spéciaux 56
Indentation 765
Indicateurs
  d'état du flux de sortie 597
  iostream 598, 599
  ostream 597
Indirection 213
Initialisation
  données membres de classe 151
  tableau 407
Instance
  de classe 144
  de modèle 643
Instanciation 643
Instructions
  #include 24
  catch(...) 713
  du préprocesseur 24
  endl 29, 595
  return 35
  std::endl 28
  switch 198
int (type de variable) 44
  type renvoyé 25
Interaction
  diagrammes 337
Interface 257
  de classes 160
  fonctions 148
  multiple (Java) 347
  publique 156
Interpréteur 6
Invariant de classe 749
Invariants() (méthode) 749
Inversion de bits 761
iostream
  fichier en-tête 24
  indicateurs 598, 605
ISO 54

J

Java 6, 11

L

Langages
  C 11
  C# 11
  C++ *Voir* C++
  compilés 6
  de modélisation *Voir* UML
  interprétés 6
  Java *Voir* Java
  procéduraux 136
Lecture d'une adresse 217
Liaison
  interne et externe 623
  liste chaînée 837
Ligne de commande, paramètres 612
Limite, écriture dans tableau 398
Linker *Voir* Éditeur de liens
Liste
  arborescence 837
  chaînée 837
  de paramètres, déclaration 269
  double 837
  nœuds 837
  simple 837
Littérales, constantes 57
Locale, variable 104
Localisation des erreurs 731
long (type de variable) 51
long int (type de variable) 44
Lvalue 69

M

Macros
  assert() 747
  contre fonctions et modèles 756
  prédéfinies 747

main() (fonction) 25, 98
  void ou int 115
Masquage
  données 9
  méthode 374
Membres
  classe agrégée *Voir* Agrégation
  et modèles 671
  statiques, données 484
Mémoire
  libération 221
  organisation 221
Méthodes
  avantages et inconvénients 391
  d'accès 145
  définition 159
  membres *Voir* Fonctions membres
  valeurs par défaut 282
mixin *Voir* Classes de fontionnalités
Modèles 321
  amis de modèles 656
  bibliothèque standard *Voir* Bibliothèque
  dynamiques 352
  et classes 652
  et macros 756
  paramétrés 643
  sous-types 349
  statiques 344
  STL *Voir* Bibliothèque
  utilisation 660
Modulo 178, 778
Mots-clés 28, 47
  class 144, 147
  const 59, 156
  continue 179
  delete 223
  enum 59
  goto 174
  inline 121, 161
  namespace 30, 625
  new 222, 677
  private 142, 150
  return 112
  signed 43
  template 644
  typedef 50
  using 630

N

namespace (mot-clé) 30
Nœuds 684
  tête 838
Noms 767
  des paramètres (fonctions) 101
Normes
  ANSI 12
  ISO 54
Notation
  hongroise 46, 767
  UML 321
Nouvelle ligne 26
Nuls
  objets 274
  pointeurs et références 250

O

Objets 9, 136
  cin *Voir* cin
  cout *Voir* cout
  création dans le tas 226
  et classes 140
  état 598
  exceptions 719
  fichiers *Voir* Fichiers
  fonctions 692
  référence 248
  tableaux 404
ofstream
  comportement par défaut 607
  objet 604
Opérateurs
  bit à bit 758
  d'extraction 580
  d'incrémentation 295
  de concaténation 746
  de redirection 17, 580
  de résolution de portée 621
  et fonctions 595
  préfixe et suffixe 302
  priorités 90
  référence 244
  restrictions 309
  sizeof() 610
  surcharge 293
  syntaxe 270
  types renvoyés 297
operator / 656
Ouverture des fichiers E/S 605

P

Paquetage 338
Paramètres
  de fonctions 107
  exemple 527
  par référence 244, 260
Passage de paramètres 108
  aux constructeurs de base 367
  et découpage données 385
Pause dans exécution 20
peek() (fonction) 593
Pile 130, 220

de données 686
pointeur de 130
structure 130
Point
d'arrêt 732
de contrôle 732
Pointeurs
adresse et valeur 215
arithmétique 411
création 214
d'instruction (pile) 129
de fonctions 492
et références 269
et suivi 275
fous 212, 233
initialisation 213
membres 507
perdus 236
rôle et fonction 210
sur fonctions membres 510
tableaux 409, 499
this 232
utilité 220
Polymorphisme 10, 379
de fonction 118
POO 9
et C++ 9, 135
Portée
dans une boucle 194
locale 132
Préfixe, surcharge 295
Préprocesseur 24
Priorité des opérateurs 777
private (mot-clé) 362
Privé(es)
données membres *Voir* Données
héritage 541
Procédures 8, 136
Programmation
commentaires 30
développement
cycle 15
environnement 13
étapes dans Visual C++ 19
niveaux d'abstraction 128
orientée objet *Voir* POO
procédurale 7
structurée 7, 136
styles 765
Programme
définition 7
déroulement 173
distribution 6
édition de liens 15
exécution 20
fichier exécutable 15
préparation 13
Projet
analyse 340
construction (étapes) 19
protected (mot-clé) 362
Prototype de fonction 100
Public/ques
dérivation 360
fonctions 143
membres 168
méthodes *Voir* Fonctions
putback() (fonction) 593

R

RAM 40
organisation 210
partitionnement 129
Rational Rose 331, 353
Récursivité 111, 122, 123
Redéfinition de méthodes 372
Redirection 579
Références
alias 246
const 268
de valeurs multiples 257
extraction d'adresses 245
nature et rôle 244
problèmes de réaffectation 247
Registres 129, 130, 221, 733
Relation
a-un 531
est-un 531
Remplissage, caractères de 598
Renvoi, variables temporaires anonymes 299
Respect casse des caractères *Voir* Casse
Retour de fonction 98
return, instructions 35
Réutilisabilité 10
RTTI 441, 444
Rvalue 69

S

Saisie de caractères 586
Scénario 334
Script 6
short (type de variable) 51
short int (type de variable) 44
Signature d'une fonction 100
sizeof() (opérateur) 43
Sorties *Voir* Impression
Source, fichier *Voir* Fichiers
stack, classe 686

Standard
  bibliothèque *Voir* Bibliothèque
  entrées et sorties 579
std 25, 28, 121, 677, 684, 686, 692
Step into 128, 302
Stockage des données en mémoire 40
strcpy() (fonction) 423
Structures 136, 168
  de la pile 130
  file d'attente 687
  pile 686
Styles de programmation 765
Suppression de tableaux 417
Surcharge
  d'addition 306
  d'insertion 566
  et fonctions amies 560
  et redéfinition 374
  fonctions 117
  suffixe 302
swap() (fonction) 108
  par référence 253
  par valeur 251
switch
  clause default 198
  instruction 766
  syntaxe 198, 200
Symboliques, constantes 58
Syntaxe
  dérivation 360
  for 188
  références 250
  tableaux 404
  while 177

T

Tableaux
  à plusieurs dimensions 406
  de caractères 421
  de chaînes de caractères 430
  de fonctions 499
  erreurs d'intervalle 401
  initialisation 402, 407
  listes chaînées 433
  nom des 415
  numérotation des éléments 396
  présentation 395
  problèmes mémoire 409
  réallocation 419
  redimensionner 418
Tabulation 28
TAD (Type abstrait de données) 476
Taille des entiers 41
Tampon
  de caractères 582
  utilisation 577
Tas
  renvoi d'une référence 272
  suppression d'objets 226
this *Voir* Pointeurs
Transtypage
  descendant 441
Tri 695
typedef 50
Types
  abstraits 463
  de commentaires 31
  de données 28
  de variables 43
  int 25
  logiques 350
  taille 42
  variables 137
  void 25

U

UC (unité centrale) 129
UML
  classes 341
  interaction entre classes 352
  relations entre classes 347
  Voir aussi Cas d'utilisation
Use case *Voir* Cas d'utilisation
using, mot-clé 28
Utilisateur, entrées 36

V

Valeurs
  intermédiaires, affichage 755
  par référence 259
  passées en paramètre 111
  récupérer pour une variable 213
  type 99
  void 112
Variables 39
  affecter une valeur 48
  caractère 54
  création 48
  et mots-clés 47
  globales 104
  intermédiaires 112
  locales 104
  nom des 45
  portée 104
  types 40
  utilisation 49
  virgule flottante 43
Virgule flottante 28

Virtuel(le)s
  destructeurs *Voir* Destructeur
  héritage *Voir* Héritage
  méthodes *Voir* Méthodes
Visibilité *Voir* Portée
Visual Basic 6
void
  type 25
  valeurs renvoyées 112
vptr 383
v-table 383

W

while
  boucle 175
write() (fonction) 610

Le Programmeur

Le langage C++

Maîtrisez rapidement les principes fondamentaux de la programmation orientée objet !

Grâce à cet ouvrage, initiez-vous rapidement au C++ en découvrant les principaux concepts du langage et de la programmation orientée objet : gestion des entrées-sorties, boucles et tableaux, modèles, etc. Ces concepts sont mis en œuvre dans de nombreux exemples de codes, qui sont analysés en détail pour vous permettre ensuite d'écrire vos propres programmes. Vous pourrez également vous exercer et tester vos connaissances avec les quizz et exercices présents dans chaque chapitre, dont vous trouverez les réponses à la fin de l'ouvrage.

Entièrement révisé pour cette nouvelle édition, ce best-seller vous guidera pas à pas dans votre apprentissage du C++.

Les codes sources du livre sont disponibles sur www.pearson.fr.

- Bien débuter en C++
- Anatomie d'un programme C++
- Variables et constantes
- Expressions et instructions
- Fonctions
- Programmation orientée objet
- Déroulement d'un programme
- Pointeurs
- Références
- Fonctions avancées
- Analyse et conception orientée objet
- Héritage
- Tableaux et chaînes
- Polymorphisme
- Classes et fonctions spéciales
- Concepts avancés d'héritage
- Les flux
- Espaces de noms
- Les modèles
- Gestion des erreurs et exceptions
- Et maintenant ?
- Mots-clés C++
- Priorité des opérateurs
- Solutions des exercices
- Étude des listes chaînées

Niveau : Débutant / Intermédiaire

Catégorie : Programmation
Configuration : Multiplate-forme

À propos de l'auteur

Jesse Liberty est l'auteur de nombreux best-sellers sur ASP.NET, C# et C++. Il a travaillé pour AT&T et Citibank avant d'intégrer les équipes de Microsoft.
Bradley L. Jones dirige de nombreux sites web dédiés à la programmation, dont CodeGuru.com, Developer.com et Javascripts.com.

ISBN : 978-2-7440-4023-8

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers 75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr